



System Programming Basics

In the first part of this book we'll discuss the basics of system programming. We'll talk about the purpose of system programming and the methods and tools used in system programming. We'll also explain the PC's basic structure and the interaction between hardware, BIOS and DOS.

What Is System Programming?

Some users, regardless if they're beginners or experienced programmers, believe system programming is a programming technique that converts a problem into a finished program. Others think system programming means developing programs for one particular computer system.

Application programming versus system programming

Although both answers are incorrect, the second is more accurate than the first. The most accurate description of system programming can be derived from the term application programming. This type of programming refers to information management and presentation within a program. This involves arranging this information into lists, etc., and processing this information. The algorithms used for this are system independent and can be defined for almost any computer.

The way this information is passed to a program, and the way the information is displayed or printed are system dependent. System programming controls any hardware that sends information to, or receives information from, the computer. However, since this information must be processed, developing programs for PCs requires both application programming and system programming. Programming hardware requires the interaction of system programming, DOS, and the ROM-BIOS (more on this later).

The Three-Layer Model

One of the most important tasks of system programming involves accessing the PC hardware. However, the access doesn't have to occur immediately, with the program turning directly to the hardware, which is similar to accessing the processor on a video card. Instead, the program can use the ROM-BIOS and DOS to negotiate hardware access. The ROM-BIOS and DOS are software interfaces, which were created specifically for hardware management.

Advantages of the DOS and BIOS interfaces

The greatest advantage of using DOS or BIOS is that a program doesn't have to communicate with the hardware on its own. Instead, it calls a ROM-BIOS routine that performs the required task. After the task is completed, the ROM-BIOS returns status information to the program as needed. This saves the programmer a lot of work, because calling one of these functions is faster than directly accessing the hardware.

There's another advantage to using these interfaces. The ROM-BIOS and DOS function interfaces keep a program isolated from the physical properties of the hardware. This is very important because monochrome graphic cards, such as the MDA and Hercules cards, must be programmed differently from color graphic cards, such as the CGA, EGA, VGA, and Super VGA. If you want a program to support all these cards, you must implement individual routines for each card, which is very time-consuming. The ROM-BIOS functions used for video output are adapted to the resident video card, so the program can call these functions without having to adapt to the video card type.

ROM-BIOS

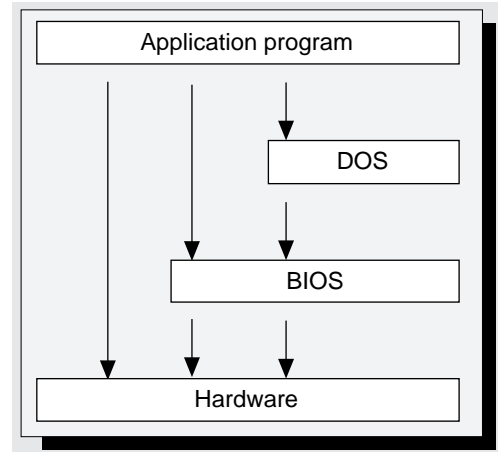
The BIOS offers functions for accessing the following devices:

- Video cards ➤ RAM (extended memory) ➤ Diskettes ➤ Hard drives
- Serial ports ➤ Parallel ports ➤ Keyboard ➤ Battery-operated realtime clock

As this illustration shows, the ROM-BIOS can be viewed as a layer overlapping the hardware.

Although you can bypass the ROM-BIOS and directly access the hardware, generally you should use the ROM-BIOS functions because they are standardized and can be found in every PC. The ROM-BIOS, as its name indicates, is in a ROM component on the computer's motherboard. The moment you switch on your computer, the ROM-BIOS is available (see Chapter 3 for more information).

The three layer model



DOS interface

Along with BIOS, DOS provides functions for accessing the hardware. However, since DOS views hardware as logical devices instead of physical devices, DOS functions handle hardware differently. For example, the ROM-BIOS views disk drives as groups of tracks and sectors, but DOS views these drives as groups of files and directories. If you want to view the first thousand characters of a file, first you must tell the ROM-BIOS the location of the file on the drive. With DOS functions, you simply instruct DOS to open a file on drive A:, C:, or whatever device, and display the first thousand characters of this file.

Access often occurs through BIOS functions used by DOS. However, sometimes DOS also accesses hardware directly, but you don't have to worry about this when you call a DOS function.

Which functions should you use?

We'll show you later how to call DOS and BIOS functions. First, however, we must determine which hardware access to use. We have the option of direct hardware programming, calling BIOS functions and calling DOS functions. First, you don't always have a choice between direct hardware programming and BIOS and DOS functions. Many tasks aren't supported by the BIOS or DOS functions. For example, if you want your video card to draw circles or lines, you won't find the appropriate functions in DOS or the BIOS. You must use direct hardware programming or purchase a commercial software library that contains this program code.

Choosing between BIOS and DOS

When either a BIOS function or a DOS function can be used, base your decision on the current situation. Use DOS functions if you want to work with files. If you want to format a diskette, you must use the appropriate BIOS functions. This is similar to displaying characters on the screen. If you want to redirect your program output to a file (e.g., DIR >LIST.TXT), you must use DOS functions. Only DOS functions automatically perform this redirection. The BIOS functions provide better control of the screen (e.g., cursor placement). So, the situation determines which function you should use.

Slowing access

However, in some instances, both the BIOS functions and DOS functions are at a disadvantage because of slow execution speed. As the number of software layers, which must be negotiated before hardware access occurs, increases, the programs become longer. If the hardware must access a program that reads a file through BIOS and DOS, a hard drive's data transfer rate can decrease a maximum of 80 percent.

This problem is caused by the way the layers are handled. Before the call can be passed to the next level, parameters must be converted, information must be loaded from internal tables, and buffer contents must be copied. The time needed for this passage is called overhead. So, as overhead increases, so does the programmer's work.

As a result, when maximum execution speed is required and direct hardware programming is relatively simple, programmers often use direct access instead of the BIOS and DOS. The best example of this is character output in text mode. Almost all commercial applications choose the most direct path to the hardware because BIOS and DOS output functions are too slow and inflexible. Direct video card access in text mode is quite easy (refer to Chapter 4 for more information), although graphic mode output offers more challenges. Later in this chapter you'll learn how to call the DOS and BIOS functions and how to directly access the hardware of the PC.

Basics Of PC Hardware

In this section we'll examine some of the basic concepts of PC architecture, which lead all the way to the system programming level. Knowing something about the hardware will make it easier to understand some of the programming problems discussed later in this book.

Birth of the PC

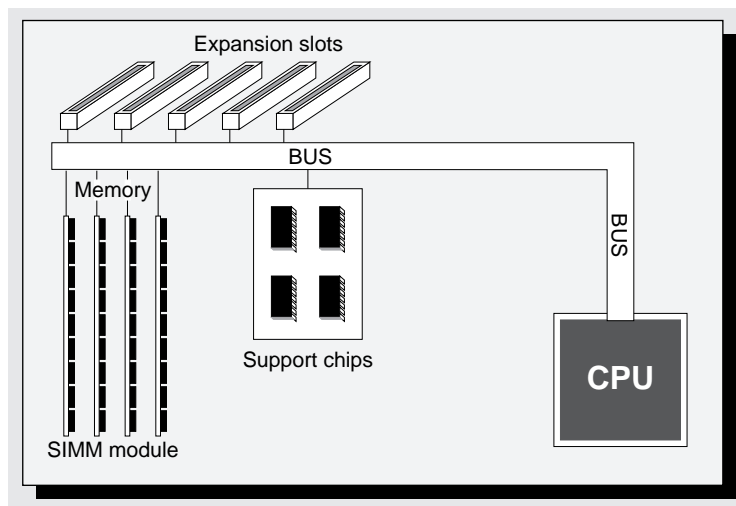
When the PC appeared on the market, much of what PC users take for granted today was inconceivable. The concept of having a flexible computer on a desktop wasn't new; companies much smaller than IBM had already introduced similar computers. IBM had just completed work on its System/23 DataMaster. However, the DataMaster was equipped with an 8085 8-bit processor from Intel, which was outdated. In 1980, the 16-bit processor was introduced and IBM began planning a new, revolutionary machine.

Choosing a processor

The 8086 processor and 8088 processors from Intel were the first representatives of the new 16-bit processors. Both had 16-bit registers. This meant they could access 1 megabyte memory addresses instead of the old 64K memory addresses. A megabyte was an unimaginable amount of memory in 1980, just as 1 Gigabyte of RAM is still unimaginable to many today.

Another reason developers were anxious to use the 8086 and 8088 processors was that many support chips already existed. Obviously this saved a lot of development time. Also, both processors were supported by an operating system and an implementation of the BASIC language, which was developed by Microsoft Corporation.

*Block diagram
of your PC's
hardware*



The developers chose the 8088 over the 8086 because, while the 8088 worked on a 16-bit basis internally, it only communicated with the outside world using an 8-bit data bus. Since the 8-bit DataMaster data bus already existed, the 8088 was the obvious choice. This bus connects the motherboard of the PC, where the processor and its support chips are resident, to the memory and the expansion boards, which are plugged into the expansion slots.

The Bus

Although the bus is vital to the operation of the computer system, the development of the PC bus represents one of the darkest moments in the history of the PC. Although IBM tried to create an open system and publish all technical information, it neglected to document the exact sequence of the bus signals, probably assuming that no one would need or want this information. However, the openness of the PC and the option of easily adding expansion boards and more hardware added to the PC's success on the market. Many users quickly took advantage of this, buying IBM expansion boards and third-party compatible boards. The PC has its entire data and address bus on the outside; the bus connects to RAM, the various expansion boards, and some support chips.

Operating the PC bus

The bus is basically a cable with 62 lines, from which data are loaded into memory by the processor, and through which data can be transported to the processor. The bus consists of the *data bus* and the *address bus*. When memory is accessed, the processor puts the address of the desired memory location on the address bus, with the individual lines indicating a binary character. Each line can be only a 0 or a 1. Together, the lines form a number that specifies the address of the memory location. The more lines that are available, the greater the maximum address and the greater the memory that can be addressed in this way. Twenty lines were available on the original address bus because with 20 bits you can address 1 megabyte of memory, which corresponds to the processor's performance.

The actual data are sent over the data bus. The first data bus was only 8 bits wide, so it could transfer only one byte at a time. If the processor wanted to discard the contents of a 16-bit register or a 16-bit value in memory, it had to split the register or value into two bytes and transfer one byte at a time.

Although theoretically this sounds simple, it's a complicated procedure. Along with the data and address buses, almost two dozen other signal lines communicate between the processor and memory. All the boards communicate with the bus. When a board takes responsibility for the specified address, it must send an appropriate signal to the processor. At this point, all the other boards separate from the rest of the communication and wait for the beginning of the next data transfer cycle.

Using expansion boards always leads to problems. This usually occurs when two boards claim the same address range or there are overlapping address ranges. The DIP switches on these boards let you specify the address range. One board must be reconfigured to avoid conflict with the other board.

As a system programmer, you'll never encounter bus signals. Bus performance usually isn't important to system programming. The bus signal timing is very important to expansion board manufacturers. Their products must follow this protocol to function in the PC. However, this is the protocol that IBM never published. So, the manufacturers must measure the signal sequences by using existing cards and then imitate those cards.

AT bus

In 1991, the IEEE (Institute of Electrical and Electronic Engineers) submitted an international standard for the AT bus. The PC bus was limited by its 8-bit width. When the AT appeared on the market, it included a 16-bit bus that was compatible with the older bus. That's why the old PC 8-bit boards can be used with the new 16-bit boards in one device. Obviously, the 16-bit boards are much faster because they can transfer the same data in half the time it would take an 8-bit board.

The address bus was expanded to 24 bits, so the AT can address 16 megabytes of memory. Also, higher clock signal speed increased bus transfer time. From 4.77 MHz on the PC, the AT speed increased to 8 MHz. However, that's as fast as the AT address bus can handle information, although Intel processor speeds have reached the 100 MHz limit. As a result, the bus is a bottleneck, through which the data will never be transferred quickly enough between memory and the processor. Modern hard drives have a higher data transfer rate than the bus.

Wait state

The wait state signals found in some expansion boards give slow boards more time to deliver data to the processor. This is also one reason why the AT bus resulted in more powerful successors like the Micro Channel bus and the EISA bus, which haven't been very successful on the market for other reasons. At first there wasn't a generic name for the AT bus. However, when competition appeared on the market, the bus was assigned the name Industry Standard Architecture bus, or ISA bus.

Problems with 16-bit boards on the AT bus

Since many 386es and 486es have an ISA bus, many problems in the PC can be traced to this bus. For example, the coexistence of 8-bit and 16-bit expansion boards within a PC causes problems if the address range for which these boards are responsible is located within any area of 128K. The problem starts at the beginning of a data transfer when a 16-bit board has to signal from a control line that it can take a 16-bit word from the bus and, unlike an 8-bit board, doesn't depend on the transfer being split into two bytes. However, the board must send this signal when it cannot even be aware the address on the data bus is intended for it and requires an answer. Of the 24 address lines that carry the desired address, only lines A17 to A23 have been correctly initialized to this point. This means the board only recognizes bits 17 to 23. These bits cover a complete 128K region, regardless of what might follow in address bits 0 to 16. So for the moment, the board only knows whether the memory address is located in the 0K-127K region, the 128K-255K region, etc.

If the 16-bit board sends the signal for a 16-bit transfer at this moment, it's speaking for all other boards within this region. They experience this in the next moment, because after address bits 0 to 16 have arrived on the bus, the intended board will be determined. If it really is the 16-bit board, no problems occur. However, if an 8-bit board was intended, the 16-bit board will simply separate from the rest of the transfer, leaving the 8-bit board by itself. However, the 8-bit board won't be able to manage the transfer because it's only set for 8-bit transfers. So, the expansion board cannot accept the data as sent.

PC BUS and VESA Local Bus

Considering the limitations of the AT bus and the inability of the EISA and MCA bus to gain market share, developers devised other bus concepts. The VESA Local bus (VL bus) was first. It was designed and publicized by the independent VESA Committee. The members of the VESA committee made it their business to define standards for graphic cards, so they didn't really have anything at all to do with PC bus design. However, graphic cards suffer from the low speed of the AT bus. That's why the VESA committee made the suggestion for a faster bus, the VESA local bus.

Unlike the EISA, MCA and PCI buses, the VL bus does not replace the ISA bus, instead, it complements it. A PC with a VL bus has a normal ISA bus and the appropriate slots for expansion cards. However, there are also one or two additional slots for cards designed for the VL bus, usually graphic cards. Only these slots are connected to the CPU through the VL bus so the other slots are left undisturbed and ISA cards can perform their work.

The VL bus is a local bus. Unlike the ISA bus, it is directly coupled to the CPU. On the one hand, that gives the bus a much higher clock speed (that of the CPU), but it also makes the bus dependent, both on the control lines of the CPU and on the clock. Along with these drawbacks, the specifications of the VESA committee aren't very well considered. As a result, the VL bus will not make the grade in the long run. Although some 486 systems often have this bus type, its popularity has fallen.

Clearly, the bus of the future remains Intel's PCI bus (Peripheral Component Interconnect). It represents a modern bus that is superior to the ISA bus not only with regard to clock speed and a larger bus width. Finally, the PCI is a bus that automatically synchronizes/tunes installed expansion cards regarding their port addresses, DMA channels and interrupts. The user no longer has to deal with this issue.

The PCI bus is independent from the CPU because a PCI bus controller is always interconnected with the CPU and the PCI bus. That makes it possible to use the PCI bus in systems that aren't based on an INTEL processor, such as an Alpha processor from DEC. In the future, the Power Macintosh with the PowerPC processor is also supposed to be equipped with a PCI bus.

PCI upgrade cards work reliably in all systems equipped with a PCI bus and can be exchanged. Only the software drivers have to be adapted to the host system, i.e., the CPU. Also, the PCI bus is not dependent on the clock of the CPU, because the PCI bus controller separates it from the CPU. If you add a newer, faster CPU to your computer, you don't have to worry about

your installed upgrade cards not being able to handle the higher clock speeds. Because the CPU and PCI bus are separate, the higher clock rates don't even affect them.

Pentium computers are almost exclusively equipped with PCI buses. The PCI bus is also becoming increasingly popular with 486 boards. Although you cannot operate an ISA card in a PCI slot, this doesn't mean you have to do without ISA cards on most systems with a PCI bus. Often a board with a PCI bus will have a "PCI to ISA bridge". This is a chip that is interconnected to the various ISA slots and the PCI bus controller. Its job is to convert signals from the PCI bus to the ISA bus. This allows you to continue running your ISA cards under the protection of the PCI bus.

Although the future belongs to the PCI bus, the ISA bus and ISA expansion boards will still be popular. Not all expansion boards require the high transfer rates made possible by the PCI bus. However, SCSI and network cards will be attached to the PCI bus in ever greater numbers in the future (especially for graphics). The speed advantage of this bus system is particularly noticeable with these cards so the hardware can keep up with the steadily increasing speed of the processor.

Controllers

Developers supplied the processor with additional chips to handle tasks the processor cannot handle on its own. These support chips are called *controllers* because they control a part of the hardware for the processor and perform many tasks. This enables the processor to concentrate on other tasks. The following pages describe these controllers and the chips initially selected by IBM. Programmable controllers are indicated in the book.

DMA controller (8237)

DMA is an acronym for Direct Memory Access. This technique transfers data directly to memory by using a device (e.g., a hard drive). This method seems to work much faster than the normal method, in which the processor prompts the hardware for each word or byte and then sends the word or byte to memory. Actually, the DMA controller's advantages are evident only with slow processors because the DMA is linked to the bus speed.

Today's processors, which work more than five times as fast as their bus, barely benefit from DMA transfer because the DMA controller in the PC is obsolete. So, the DMA controller cannot even be used for one of the most interesting areas of programming, which is moving large amounts of data from conventional RAM to video RAM (RAM on the video card). This chip is still found in all PCs although it isn't used for its original purpose, which is data transfer between disk drives and memory. ATs have two DMA controllers.

The PC includes DRAM (dynamic RAM) instead of SRAM (static RAM). DRAMs lose their contents unless the system continually refreshes the RAM. The DMA controllers in AT systems perform this RAM refresh instead of the processors.

Interrupt controller (8259)

The interrupt controller is important for controlling external devices, such as the keyboard, hard drive or serial port. Usually the processor must repeatedly prompt a device, such as the keyboard, in short intervals to react immediately to user input and pass this input to the program currently being executed. However, this continual prompting, also called *polling*, wastes processor time because the user doesn't press a key as often as the processor polls the keyboard. However, the less often the processor prompts the keyboard, the longer it takes until a program notices that a key has been pressed. This obviously defeats the purpose, since the system is supposed to react promptly.

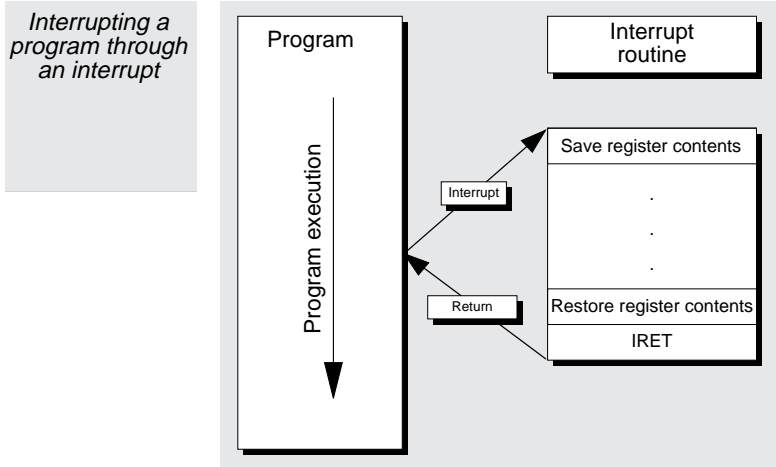
Hardware interrupt

The PC takes another route. Instead of the processor repeatedly prompting the devices, the devices report activity to the processor. This is an example of a hardware interrupt, because at that exact moment the processor interrupts the execution of the current program to execute an interrupt handler. This interrupt handler is a small routine, usually provided by the BIOS, that deals with the event that triggered the interrupt. After the routine ends, the processor continues executing the interrupted program as though nothing happened. This means the processor is called only when something actually happens.

However, the process of triggering an interrupt, halting program execution, and calling the interrupt handler takes a long time. Expansion board and support chip interrupt requests are sent to the interrupt handler first, instead of to the processor. The PC

has several interrupt lines, each connected to a device. Each of these devices could trigger an interrupt over its line simultaneously. Because the processor can only process one interrupt at a time, priorities must be defined so the incoming interrupt requests are handled according to their priority. The interrupt controller is responsible for determining priority.

The interrupt controller in a PC/XT can process up to eight interrupt sources, which enables it to handle eight interrupt requests simultaneously. Since this isn't sufficient for an AT, two interrupt controllers are coupled on the AT. Together they can process up to 15 interrupt requests simultaneously. For more information about hardware interrupts, refer to the "Interrupts" section.



Programmable peripheral interface (8255)

This chip connects the processor to peripheral devices, such as the keyboard and speaker. It acts only as a mediator, which is used by the processor to pass given signals to the desired device. (Refer to Chapter 13 for more information on this chip and how it's used to make musical sounds.)

The clock (8248)

If the microprocessor is the brain of the computer, then the clock could be considered the heart of the computer. This heart beats several million times a second (about 14.3 MHz) and paces the microprocessor and the other chips in the system. Since almost none of the chips operate at such high frequencies, each support chip modifies the clock frequency to its own requirements.

The timer (8253)

The timer chip can be used as a counter and timekeeper. This chip transmits constant electrical pulses from one of its output pins. The frequency of these pulses can be programmed as needed, and each output pin can have its own frequency. Each output pin leads to another component. One line goes to the audio speaker and another to the interrupt controller. The line to the interrupt controller triggers interrupt 8 at every pulse, which advances the timer count.

CRT controller (6845)

Unlike the chips we've discussed so far, the CRT (Cathode Ray Tube) controller is separate from the PC's motherboard (main circuit board). This chip is located on the video card, which is mounted in one of the computer's expansion slots. Originally the controller was a Motorola 6845 model controller, which was used on the CGA and MDA video cards first released by IBM. The later EGA and VGA cards superseded these cards because of their more powerful processors. Even though these new chips are no longer compatible with the original Motorola controllers, this doesn't affect the processor. Unlike the other support chips, the processor doesn't come directly into contact with the CRT controller. The ROM-BIOS is specially adapted to working with the CRT controller, which relieves the processor of the task (see Chapter 4 for more information about programming video cards).

Disk controller (765)

This chip is also usually located on an expansion board. It's addressed by the operating system and controls disk drive functions. It moves the read/write head of the disk drive, reads data from the diskette, and writes data to the diskette. Similar to the CRT controller, the disk controller is addressed by the ROM-BIOS instead of by the processor. The first PCs used a cassette drive interface instead of a disk drives. IBM assumed that this would be the preferred storage device. However, IBM stopped using the cassette interface when disk drives soon became available. Data storage on a disk drive is much safer, faster and more convenient than on a cassette. (See Chapter 14 for more information on diskettes, hard drives and their controllers.)

The math coprocessors (8087/80287/80387/80487)

Until the 80486 was released, Intel processors weren't able to work with floating point numbers. They could only process whole numbers. Depending on the bit width, integers cover a value range of 0 to 255 (8 bit), 0 to 65535 (16 bit) or 0 to 429624976 (32 bit), while floating point numbers cover the range of real numbers. That's why floating point numbers are used wherever it's necessary to calculate with real numbers, for example in a spreadsheet or CAD program. While floating point numbers can be represented with the help of integers and it is possible to base floating point arithmetic on integers via software, calculating floating point numbers is much faster when done directly in the hardware.

That is why Intel offered special math coprocessors that could be plugged into a free socket on the motherboard, next to the CPU. They were adapted to the successors of the Intel 8088, from generation to generation. There is a math coprocessor for each Intel processor up to the 486 SX. The 486 DX and the various versions of the Pentium chip have this coprocessor built in, so they are able to execute floating point calculations without adding a special coprocessor. However, there is one requirement. The software must really make use of the appropriate machine language commands for floating point arithmetic.

We won't discuss programming a coprocessor in this book because this involves normal assembly language processing instead of system programming. (Refer to Chapter 16 for more information about coprocessors.)

Memory layout

The first PCs included 16K of memory which could be upgraded to 64K on the motherboard. IBM also sold memory expansion boards containing 64K of memory which could be inserted in one of the five expansion slots. You could upgrade your PC to 256K of memory by installing up to three of these boards. This was considered a lot of memory in 1981.

The PC developers defined a memory layout that allowed RAM expansion to 640K. Along with the RAM expansion, they also planned for additional video RAM, additional ROM-BIOS, and some ROM expansions in the 1 megabyte address space of the 8088 processor.

Whether RAM or ROM is in a given memory location doesn't matter to the processor, except that ROM locations cannot be written. The processor can also address memory locations that don't exist physically. Although the processor can manage up to 1 megabyte of memory, this doesn't guarantee that a RAM or ROM component exists behind every memory address.

As the following table shows, this memory layout is based on 64K segments because the 8088 and its successors manage memory in blocks of this size (more on this in Chapter 12). Sixteen of these blocks comprise an address space of 1 megabyte.

Division of PC RAM		
Block	Address	Contents
15	F000:0000 - F000:FFFF	ROM-BIOS
14	E000:0000 - E000:FFFF	Free for ROM cartridges
13	D000:0000 - D000:FFFF	Free for ROM cartridges
12	C000:0000 - C000:FFFF	additional ROM-BIOS
11	B000:0000 - B000:FFFF	Video RAM
10	A000:0000 - A000:FFFF	Additional video RAM (VGA/EGA)
9	9000:0000 - 9000:FFFF	RAM from 576K to 640K
8	8000:0000 - 8000:FFFF	RAM from 512K to 576K
7	7000:0000 - 7000:FFFF	RAM from 448K to 512K
6	6000:0000 - 6000:FFFF	RAM from 384K to 448K
4	5000:0000 - 5000:FFFF	RAM from 320K to 384K
5	4000:0000 - 4000:FFFF	RAM from 256K to 320K
3	3000:0000 - 3000:FFFF	RAM from 192K to 256K
2	2000:0000 - 2000:FFFF	RAM from 128K to 192K
1	1000:0000 - 1000:FFFF	RAM from 64K to 128K
0	0000:0000 - 0000:FFFF	RAM from 0K to 64K

The first 10 memory segments are reserved for conventional memory, limiting its size to 640K. Memory segment 0 is important because it contains important data and operating system routines.

Memory segment A follows conventional memory. This segment indicates an EGA or VGA card and contains additional video RAM for generating the various graphics modes supported by these cards.

Memory segment B is reserved for a Monochrome Display Adapter (MDA) or Color/Graphics Adapter (CGA). They share the same segment of video RAM. The monochrome card uses the lower 32K and the color card uses the upper 32K. Each video card only uses as much memory as it needs for the display. The MDA uses 4K while the CGA card uses 16K.

The next memory segment contains ROM beginning at segment C. Some computers store the BIOS routines that aren't part of the original BIOS kernel at this location. For example, the XT uses these routines for hard drive support. Since this location isn't completely utilized, this memory range may be used later to store BIOS routines supporting hardware extensions.

ROM cartridges

Segments D and E were originally reserved for ROM cartridges, but they were never properly used. Today this range is used either for additional RAM or EMS memory (see Chapter 12 for more information).

Segment F contains the actual BIOS routines, the original system loader, and the ROM BASIC available on early PCs.

Following this memory layout

The PC hardware isn't limited to any particular memory layout, including IBM's. However, IBM set the standard with its first PC, and suppliers still follow this standard. This usually affects software because the BIOS and DOS have adapted to the locations of certain memory areas (e.g., video RAM). Every software product on the market also complies with IBM's memory structure.

After the PC

Although the original IBM PC wasn't the last development in the PC world, it did establish a series of basic concepts, including the BIOS functions, the memory layout, and the interaction between the processor and the support chips.

However, the XT and the AT brought a few small changes to these concepts. The XT, released in 1983, had the first hard drive with a 10 megabyte capacity. This upgrade barely affected the total system, except the C segment was given an additional hard drive ROM, which added some ROM-BIOS functions for hard drive access.

The AT

The AT (Advanced Technology) computer was released in 1984, only one year after the XT. The most significant improvement involved the processor because developers used the Intel 80286 instead of the 8088. This processor finally gave the PC a 16-bit data bus. So, memory accesses no longer had to be divided into two bytes, as long as the memory and expansion board cooperated. Also, the address lines of the bus were increased from 20 to 24 bits because the 80286 could manage 24-bit addresses, which allowed it to address a memory range of 16 megabytes.

Disk drives

The AT doubled the hard drive capacity to 20 megabytes and introduced the 5.25" HD (high density) disk drive with a capacity of 1.2 megabytes. This disk drive is still used today. Also, the AT had a battery operated realtime clock, which finally made it possible for the clock to continue running even after the computer was switched off. The AT also increased the number of DMA controllers and interrupt controllers to two each.

A few new ROM-BIOS functions, such as functions for accessing the battery operated realtime clock, supported the new hardware.

Although the AT provided many improvements, it signaled the beginning of a trend that favors the current version instead of creating solutions for future upgrades. For example, "downward compatibility" in protected mode (an operating mode that separated the 80286 from its predecessors) wasn't widely used until the 80386 and Windows 3.0 were introduced.

When the 80286 appeared, preparations hadn't been made for protected mode. DOS, BIOS, and software avoided supporting this mode. Users continued working in real mode, in which the 80286 acts like a glorified 8088, performing at a fraction of its total capacity. Unfortunately, this is still happening today; real mode will probably be used until the switch to Windows NT and OS/2.

PS/2

After the AT, IBM attempted to set another standard with its PS/2 systems. These systems were successful mainly because of an improved bus system called the Micro-Channel Architecture (MCA). However, IBM kept the architecture of the new bus secret. It provided the information needed for building expansion cards only to hardware manufacturers that paid the licensing fees. This resulted in a limited supply of expansion boards for a system that wouldn't accept any AT boards. ISA boards cannot be used in systems with an MCA bus because the MCA bus has an entirely different line capacity.

No standards after the AT

Many companies began offering less expensive (and sometimes better) alternatives to the AT and PS/2. Companies like Compaq, which released laptop computers and an AT that had an 80386 processor, kept PC technology moving forward.

However, no company could fill the gap that was left by IBM when it dropped in the market. Once the PC market became fragmented, none of the companies had the power to define new hardware/software standards and push them onto the market. After a few years, committees met to set hardware standards (e.g., the Super VGA standard) that improved system and software compatibility.

After the AT, a new PC based on the ISA bus wasn't defined. So, systems with 80386 or 80486 processors are still generically referred to as ATs because they're based on the technology introduced by IBM when the AT was released.

The Processor

You don't have to become a professional assembly language programmer to understand system programming. You can also use high level languages, such as BASIC, Pascal, or C, for system programming. However, you must understand some concepts of the processor that are important in system programming. These concepts, which overlap into high level language programs, include the processor register, memory addressing, interrupts, and hardware access.

Although these principles haven't changed much since the 8088 was introduced, this chip is in its fifth generation and has capabilities that were unheard of ten years ago. However, these changes relate to the processor's speed instead of its fundamental concept.

The PC's brain

Let's discuss the family of Intel PC processors. The *microprocessor* is the brain of the PC. It understands a limited number of assembly language instructions and processes or executes programs in this assembly language. These instructions are very simple and can't be compared to commands in high level languages, such as BASIC, Pascal, or C. Commands in these languages must be translated into numerous assembly language instructions the PC's microprocessor can then execute. For example, displaying text with the BASIC PRINT statement requires the equivalent of several hundred assembly language instructions.

Assembly language instructions are different for each microprocessor used in different computers. The terms Z/80, 6502, or 8088 assembly language (or machine language) refer to the microprocessor being programmed.

Intel's 80xx series

The PC has its own family of microprocessor chips, which were designed by the Intel Corporation. The following figure shows the Intel 80xx family tree. Your PC may contain an 8086 processor, an 8088 processor (used in the PC/XT), an 80186 processor, an 80286 processor (used in the AT), or even an 80386 processor microprocessor. The first generation of this group (the 8086) was developed in 1978. The successors of the 8086 were different from the original chip. The 8088 is actually a step backward because it has the same internal structure and instructions of the 8086, but is slower than the 8086. The reason for this is the 8086 transfers 16 bits (2 bytes) between memory and the microprocessor simultaneously. The 8088 is slower since it transfers only 8 bits (1 byte) at a time.

The other microprocessors of this family are improved versions of the 8086. The 80186 provides auxiliary functions. The 80286 has additional registers and extended addressing capabilities. However, the 80286's greatest innovation is protected mode (see Chapter 33 for more information). DOS doesn't support protected mode.

The 80386 followed the 80286, and marks a great leap forward in performance. However, it's already outdated, and you will hardly find 386es on the market any more. This processor has advanced protected mode and 32-bit registers. Like protected mode, DOS doesn't support these registers. The 80386 includes SX and DX versions, which differ in clock frequency and data bus width. The SX works with a 16-bit data bus, while the DX can transfer an entire 32-bit word at one time.

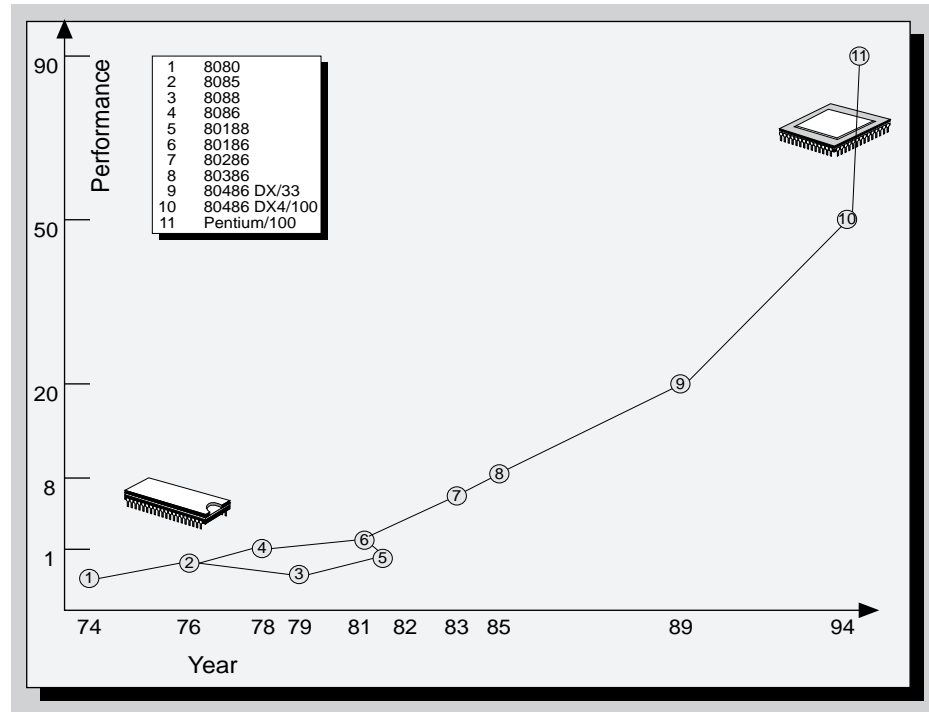
The 80486 (often simply called "486") is no longer the most advanced processor. It remains, however, very popular and sells in high numbers. It differs from the 80386 because it includes the 80387 math coprocessor, a code cache, and faster processing of many assembly language instructions. However, the 486 also maintains downward compatibility with the 8086.

The Pentium is today's most advanced processor. The main improvement in the Pentium compared to the 486 is the internal processing speed. In specific situations, this processor is able to process two sequential commands simultaneously, provided the second command doesn't depend on the result of the first command.

The name of the processor, Pentium, is also new. Users were expecting the 80586. Intel preferred to break with tradition, because names such as 8088 or 80586 cannot be protected by copyright. Other chip manufacturers took advantage of this to sell Intel compatible processors under similar names. Intel decided to take the wind out of the competition's sails and came up with "Pentium", which is protected by copyright.

No one knows yet whether the Pentium will be followed by the "Hexium", but we can start looking forward to the next generation of Intel processors, which will be introduced in 1995.

The Intel 80xx processor family

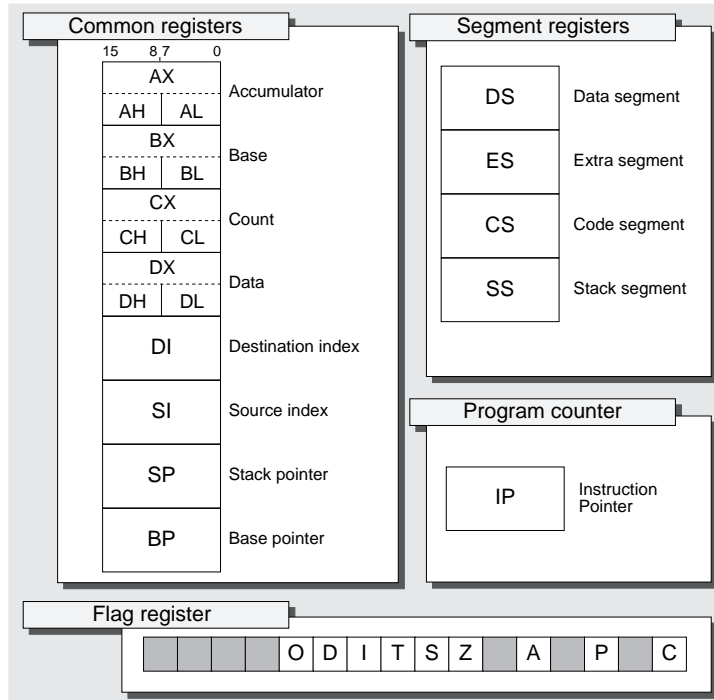


Processor registers

Registers are memory locations within the processor itself instead of in RAM. These registers can be accessed much faster than RAM. Registers are also specialized memory locations. The processor performs arithmetic and logical operations using its registers. The processor registers are important for system programming because the flow of information between a program and the DOS and BIOS functions that call this program occurs through these registers.

From a system programming viewpoint, nothing has changed in registers since the 8086. This is because the BIOS and DOS were developed in connection with this processor, so they only support this processor's 16-bit registers. The 32-bit registers of an 80386 and i486 cannot be used in system programming under DOS. We'll discuss only 8088 registers, which apply to all later chips.

*8088 registers
also apply to
later processors*



All registers are 16 bits (2 bytes) in size. If all 16 bits of a register contain a 1, the result, which is the decimal number 65535, is the largest number that can be represented within 16 bits. So, a register can contain any value from 0 to 65535 (FFFFH or 111111111111111b).

Register groupings

As the illustration above shows, registers are divided into four groups: common registers, segment registers, the program counter and the flag register. The different register assignments are designed to duplicate the way in which a program processes data, which is the basic task of a microprocessor.

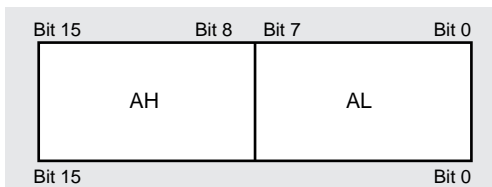
The disk operating system and the routines stored in ROM use the common registers extensively, especially the AX, BX, CX, and DX registers. The contents of these registers tell DOS what tasks it should perform and which data to use for execution.

These registers are affected mainly by mathematical (addition, subtraction, etc.) and input/output instructions. They are assigned a special position within the registers of the 8088 because they can be separated into two 8-bit (1 byte) registers. Each common register usually contains three registers: a single 16-bit register and two smaller 8-bit registers.

Common registers

The common registers are important for calling DOS and BIOS functions and are used to pass parameters to a particular function that needs these parameters for execution. These registers are also influenced by mathematical operations (addition, subtraction, etc.), which are the central focus of all software activities at processor level. Registers AX, BX, CX, and DX have

*8088 registers
also apply to
later processors*



a special position within this set of registers, because they can be divided into two 8-bit registers. This means that each of these registers consists of three registers, one big 16-bit register and two small 8-bit registers.

The small registers have H (high) and L (low) designators. So, the 16-bit AX register may be divided into an 8-bit AH and an 8-bit AL register. The H and the L register designators occur in such a way the L register contains the lower 8 bits (bit 0 through 7) of the X register, and the H register contains the higher 8 bits (bits 8 through 15) of the X register. The AH register consists of bits 8-15 and the AL register consists of bits 0-7 of the AX register.

However, the three registers cannot be considered independent of each other. For example, if bit 3 of the AH register is changed, then the value of bit 11 of the AX register also changes automatically. The values change in both the AH and the AX registers. The value of the AL register remains constant since it is made of bits 0-7 of the AX register (bit 11 of the AX register doesn't belong to it). This connection between the AX, the AH, and the AL register is also valid for all other common registers and can be expressed mathematically.

You can determine the value of the X register from the values of the H and the L registers, and vice versa. To calculate the value of the X register, multiply the value of the H register by 256 and add the value of the L register.

Example: The value of the CH register is 10 and the value of the CL register is 118. The value of the CX register results from $CH*256+CL$, which is $10*256+118 = 2678$.

By specifying register CH or CL, you can read or write an 8-bit data item from or to any memory location. Read or write a 16-bit data item from or to a memory location by specifying register CX..

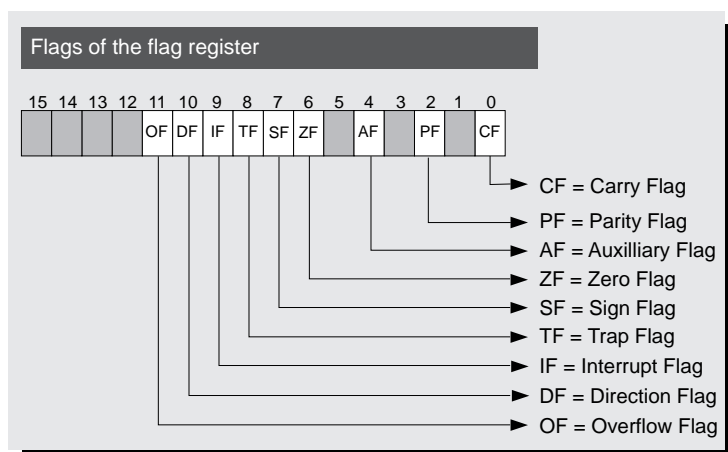
Flag register

Besides common registers, segment registers and the flag register are an important part of system programming. The flag register communicates between consecutive assembly language instructions by storing the status of mathematical and logical operations. For example, after using the carry flag to add two 16-bit registers, a program can determine whether the result is greater than 65,535 and thus present it as a 32-bit number. The sign, zero, and overflow bits perform similar tasks and can be used after two registers have been compared to establish whether the value of the first register is greater than, less than or equal to the value of the second register.

Only the carry flag and zero flag are important for system programming from high level languages. Most DOS and BIOS functions use these flags to indicate errors for insufficient memory or unknown filenames (see Chapter 2 for information on accessing these flags from high level languages).

Memory addresses

How the processor generates memory addresses is especially important for system programming, because you must constantly pass buffer addresses to a DOS or BIOS function. In these instances, you must understand what the processor is doing. The 8088 and its descendants use a complicated procedure. So that you'll understand this procedure, we'll discuss the origins of the 8086.



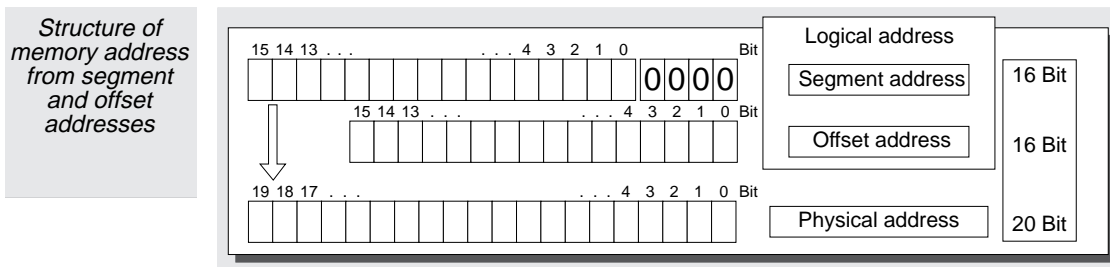
One of the design goals of the 8088 was to provide an instruction set that was superior to the earlier 8-bit microprocessors (6502, Z/80, etc.). Another goal was to provide easy access to more than 64K of memory. This was important because increasing processor capabilities allows programmers to write more complex applications, which require more memory. The designers of the 8088 processor increased the memory capacity or address space of the microprocessor (more than 16 times) to one megabyte.

Address register

The number of memory locations that a processor can access depends on the width of the address register. Since every memory location is accessed by specifying a unique number or address, the maximum value contained in the address register determines the address space. Earlier microprocessors used a 16-bit address register, which enables users to access addresses from 0 to 65535. This corresponds to the 64K memory capacity of these processors. To address one megabyte of memory, the address register must be at least 20 bits wide. At the time the 8088 was developed, it was impossible to use a 20-bit address register, so the designers used an alternate way to achieve the 20-bit width. The contents of two different 16-bit numbers are used to form the 20-bit address.

Segment register

One of these 16-bit numbers is contained in a segment register. The 8088 has four segment registers. The second number is contained in another register or in a memory location. To form a 20-bit number, the contents of the segment register are shifted left by 4 bits (thereby multiplying the value by 16) and the second number is added to the first.



Segment and offset addresses

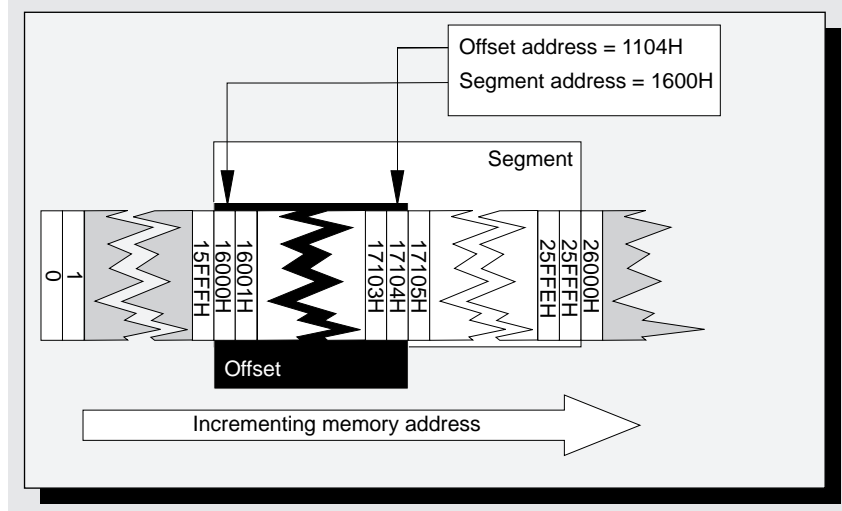
These addresses are the segment address and the offset address. The segment address, which is formed by a segment register, indicates the start of a segment of memory. When the address is created, the offset address is added to the segment address. The offset address indicates the number of the memory location within the segment whose beginning was defined by the segment register. Since the offset address cannot be larger than 16 bits, a segment cannot be larger than 65,535 bytes (64K).

Let's assume the offset address is always 0 and the segment address is also 0 at first. In this case, you receive the address of memory location 0. If the segment address is increased to 1, you receive the address of memory location 1 instead of memory location 16. This happens because the segment address is multiplied by 16 when addresses are formed.

If you continue incrementing the segment address, you'll receive memory addresses of 32, 48, 64, etc., if the offset address continues to be 0. According to this principle, the maximum memory address is 1 megabyte when the segment address reaches 65535 (FFFFH), which is its maximum value. However, if you keep the segment address constant and increment the offset address instead, the segment address will quickly become the base address for a memory segment from which you can reach a total of 65,536 different memory locations. Each memory segment contains 64K. The offset address represents the distance of the desired memory locations from the beginning of the segment.

Although the individual memory segments are only 16 bytes apart, they contain 64K. So they obviously overlap in memory. Because of this, a memory address, such as 130, can be represented in various ways by using segment and offset addresses.

Structure of memory address from segment and offset addresses



For example, you could specify 0 as the segment address and 130 as the offset address. It's also possible to specify 1 as the segment address and 114 as the offset address or 2 as the segment address and 98 as the offset address, etc. These overlapping segments are easy to use. When you specify an address you can choose the combination of segment address and offset address yourself. You must obtain the desired address by multiplying the segment address by 16 and adding the offset address to it; everything else is unimportant.

A segment cannot start at every one of the million or so memory locations. Multiplying the segment register by 16 always produces a segment address that is divisible by 16 (i.e., it's not possible for a segment to begin at memory location 22).

Segmented address

The *segmented address* results from the combined segment and offset addresses. This segmented address specifies the exact number of the memory location that should be accessed. Unlike the segmented address, the segment and the offset addresses are *relative addresses* or *relative offsets*.

Combining the segment and offset addresses requires special address notation to indicate a memory location's address. This notation consists of the segment address, in four-digit hexadecimal format, followed by a colon, and the offset address in four-digit hexadecimal format. For example, in this notation a memory location with a segment address of 2000H and an offset address of AF3H would appear as "2000:0AF3". Because of this notation, you can omit the H suffix from hexadecimal numbers.

The segment register for program execution

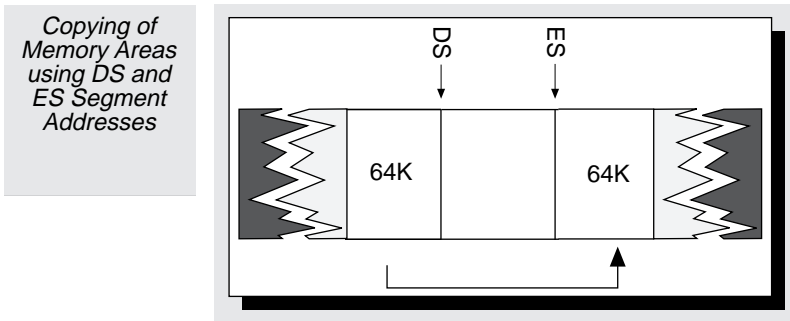
The 8088 contains four important segment registers for the execution of an assembly language program. These registers contain the basic structure of any program, which consists of a set of instructions (code). Variables and data items are also processed by the program. A structured program keeps the code and data separate from each other while they reside in memory. Assigning code and data their own segments conveniently separates them. These segment registers are as follows:

- CS The CS (Code Segment) register uses the IP (Instruction Pointer) register as the offset address. Then it determines the address at which the next assembly language instruction is located. The IP is also called the Program Counter. When the processor executes the current instruction, the IP register is automatically incremented to point to the next assembly language instruction. This ensures the instructions are executed in the proper order.
- DS Like the CS register, the DS (Data Segment) register contains the segment address of the data the program accesses (writing or reading data to or from memory). The offset address is added to the content of the DS register and may be contained in another register or may be contained as part of the current instruction.

SS The SS (Stack Segment) register specifies the starting address of the stack. The stack acts as temporary storage space for some assembly language programs. It allows fast storage and retrieval of data for various instructions. For example, when the CALL instruction is executed, the processor places the return address on the stack. The SS register and either the SP or BP registers form the address that is pushed onto the stack.

When accessing the stack, address generation occurs from the SS register in conjunction with the SP or BP register.

ES The last segment register is the ES (Extra Segment) register. It's used by some assembly language instructions to address more than 64K of data or to transfer data between two different segments of memory.



With the help of the ES register, however, it's possible to leave the DS register on the memory segment of the source area while referencing the target area using the ES memory segment. The 8088 and its descendants even have assembly language instructions that can copy an entire buffer by assuming, before their execution, the segment address of the start area has been loaded into the DS register and the segment address of the target area has been loaded into the ES register. To copy the instructions also need the start of both areas within their memory segments. They expect the start of the source area in the SI register and the start of the target area in the DI register. Expressed in the notation introduced earlier, these instructions copy data from DS:SI to ES:DI.

Overlapping segments

As the illustration on the following page shows, two segment registers can specify areas of memory that overlap or are completely different from each other. Usually a program doesn't require a full 64K segment for storing code or data. So, you can conserve memory by overlapping the segments. For example, you can store data, which immediately follows the program code, by setting the DS and CS registers accordingly.

NEAR and FAR pointers

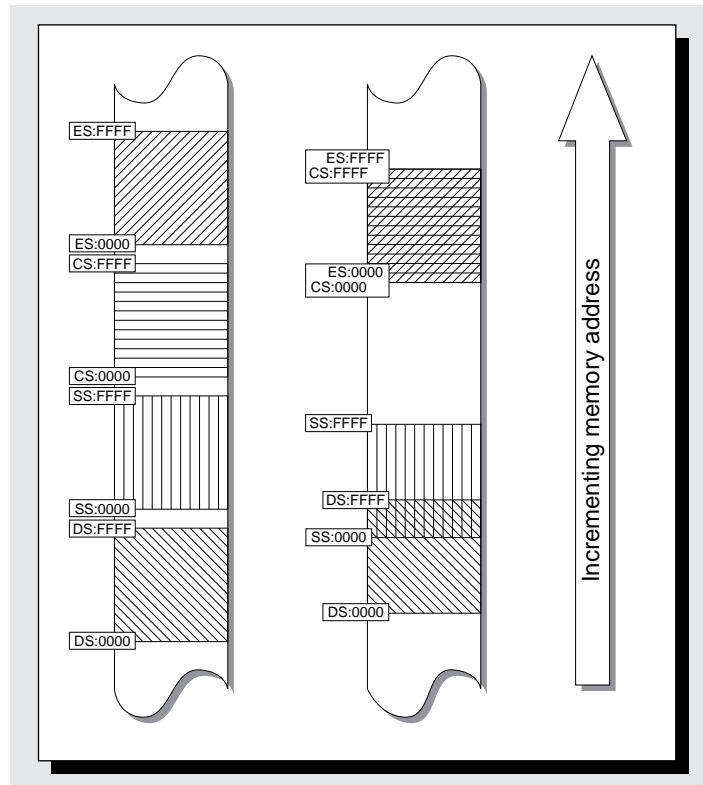
The numbers we've been calling memory addresses are called pointers in high level languages. A pointer in the Pascal or C language receives the addresses of the objects referenced by the pointers. If these addresses change location in memory, the pointers also change. The two types of pointers are NEAR pointers and FAR pointers. NEAR pointers specify the offset address of an object and are only 16 bits wide. Memory cannot be accessed without a segment address. So the compiler prepares the segment address, which it automatically loads, to the appropriate segment register when accessing the object. Because of this, NEAR pointer access is only possible for variables within the 64K segment created by the compiler.

FAR pointers consist of a segment address and an offset address, so they are saved as two words. The low word receives the offset address and the high word receives the segment address. In Turbo Pascal, pointers are VAR, while in C their type depends on the memory model (see Chapter 2 for more information about pointers).

Data types and their storage

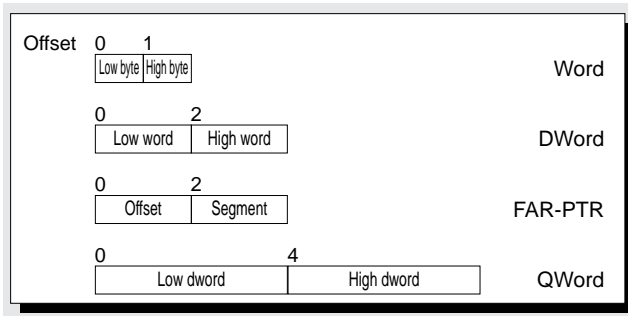
Bytes and words aren't the only data types you'll encounter in system programming. You'll frequently encounter DWORDs (double words), which are used when the 16 bits of one word aren't enough to store a number. For example, this applies to the internal BIOS clock, which exceeds the 16-bit level of 65535 after a little more than ten hours.

Non-overlapping
(left) and
overlapping
(right) segments



The members of the Intel 80xxx family place DWORDs in memory so the low word (bits 0 to 15) precedes the high word (bits 16 to 31). This procedure is referred to as the *little endian* format. This is different than the *big endian* format, which reverses the order and is used by processors of the Motorola 68000 family (e.g., the Apple Macintosh). The little endian principle also applies to word storage, in which the low word is placed in front of the high word. Even with QWORDS (4 words), which are used by the numerical coprocessor, the low-order DWORD (bits 0 to 31) is stored in front of the high-order DWORD (bits 32 to 63). Then, within these two DWORDs, the high word is placed in front of the low word, etc. The following illustration demonstrates this principle:

Storing different
data types in
little endian
format



Ports

Ports represent interfaces between the processor and the other system hardware. A port is similar to an 8-bit wide data input or output connected to a specific piece of hardware. It has an assigned address with values ranging from 0 to 65,535. The processor uses the data bus and address bus to communicate with the ports. If the processor needs to access a port, it transmits a port control signal. This signal instructs the other hardware the processor wants to access a port instead of RAM.

Although ports have addresses that are also assigned to memory locations in RAM, these addresses aren't related to the memory locations. The port address is placed on the lowest 16 bits of the address bus. This instructs the system to transfer the eight bits of information on the data bus to the proper port. The hardware connected with this port receives the data and responds accordingly. The 80(x)xx processor series has two instructions that control this process from within a program. The IN instruction sends data from the processor to a port and the OUT instruction transfers data from a port into the processor. Each hardware device is responsible for an area of port addresses. Therefore, conflicts between expansion boards that allocate the same port address area often occur. So, most expansion boards have DIP switches for setting the port address to which the board will respond. This helps avoid conflicts with other boards.

The system can set the port address of a certain hardware device. Since this address isn't a constant value, port addressing is similar for the PC, XT, and AT. Although there are only a few differences between the PC and XT, there are many differences between the PC and AT. The following table shows the port addresses of individual chips in each system.

Component	PC/XT	AT
DMA controller (8237A-5)	000-00F	000-01F
Interrupt controller (8259A)	020-021	020-03F
Timer	040-043	040-05F
Programmable Peripheral Interface (PPI 8255A-5)	060-063	none
Keyboard (8042)	none	060-06F
Realtime clock (MC146818)	none	070-07F
DMA page register	080-083	080-09F
Interrupt controller 2 (8259A)	none	0A0-0BF
DMA controller 2 (8237A-5)	none	0C0-0DF
Math coprocessor	none	0F0-0F1
Math coprocessor	none	0F8-0FF
Hard drive controller	320-32F	1F0-1F8
Game port (joysticks)	200-20F	200-207
Expansion unit	210-217	none
Interface for second parallel printer	none	278-27F
Second serial interface	2F8-2FF	2F8-2FF
Prototype card	300-31F	300-31F
Network card	none	360-36F
Interface for first parallel printer	378-37F	378-37F
Monochrome Display Adapter and parallel interface	3B0-3BE	3B0-3BF
Color/Graphics Adapter	3D0-3DF	3D0-3DF
Disk controller	3F0-3F7	3F0-3F7
First serial interface	3F8-3FF	3F8-3FF

Interrupts

In the "Basics of PC Hardware" section in this chapter we explained that interrupts are mechanisms that force the processor to briefly interrupt the current program and execute an interrupt handler. However, this is only one aspect of interrupts. They are also important for controlling the hardware, and act as the main form of communication between a program and the BIOS and DOS functions.

Software interrupts

Software interrupts call a program, with a special assembly language instruction, to execute a DOS, BIOS, or EMS function. The program execution isn't really interrupted; the processor views the called function as a subroutine. After the subroutine executes, the processor continues with the calling program.

To call a DOS or BIOS function using a software interrupt, only the number of the interrupt, from which the routine can be reached, is needed. The caller doesn't even need to know the address of the routine in memory. These routines are standardized. So, regardless of your DOS version, you know that by calling interrupt 21H you can access DOS functions. The processor calls the interrupt handler using the interrupt vector table, from which the processor takes the addresses of the desired function. The processor uses the interrupt number as an index to this table. The table is set during system startup so the various interrupt vectors point to the ROM-BIOS.

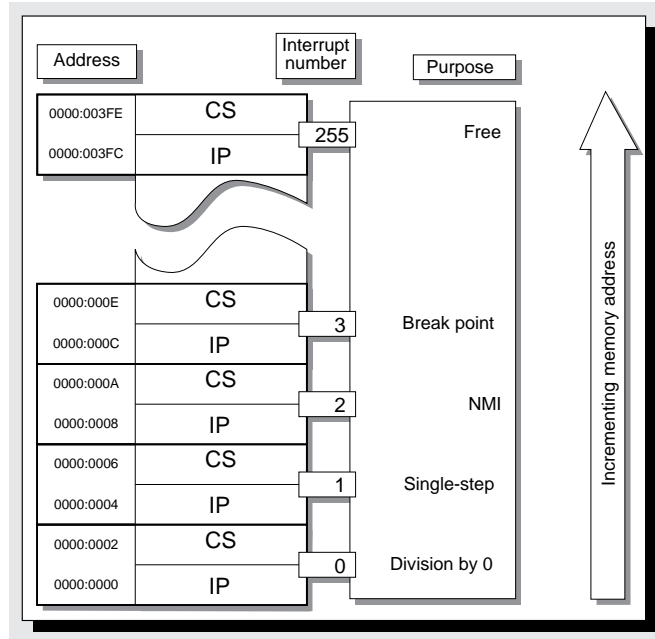
This illustrates the advantage of using interrupts. A PC manufacturer who wants to produce an IBM compatible PC cannot copy the entire ROM-BIOS from IBM. However, the manufacturer is allowed to implement the same functions in its ROM-BIOS, even if the BIOS functions are coded differently from within. So, the BIOS functions are called using the same interrupts that IBM uses and expect parameters in the same processor registers. But the routines that provide the functions are organized differently than the routines provided by IBM. Other advantages of using interrupts are described in Chapter 2. First, let's look at the interrupt vector table, which represents the key to calling the interrupts.

Interrupt vector table

So far we've discussed a single interrupt and a single interrupt routine. Actually, the 8088 has 256 possible interrupts numbered from 0 to 255. Each interrupt has an associated *interrupt routine* to handle the particular condition. To organize the 256 interrupts, the starting addresses of the corresponding interrupt routines are arranged in the *interrupt vector table*.

When an interrupt occurs, the processor automatically retrieves the starting address of the interrupt routine from the interrupt vector table. The starting address of each interrupt routine is specified in the table in terms of the offset address and segment address. Both addresses are 16 bits (2 bytes) wide. So each table entry occupies 4 bytes. The total length of the table is 256x4 or 1024 bytes (1K). Because the interrupt vector table is in RAM, any program can change it. However, TSR programs and device drivers use the table the most (see Chapter 35).

Starting addresses of the interrupt routines are arranged in the interrupt vector table



The following table shows the addresses of the various interrupt vectors, as well as the utilities from which they can be reached. This layout applies to all PCs and is an essential component of the PC standard. A program that uses these interrupts will find these utilities on all PCs. Most of these interrupts and their functions are mentioned throughout this book.

Many of these interrupt vectors are only allocated when the corresponding hardware has also been installed. For example, this applies to interrupt 33H (mouse driver functions) and interrupt 5CH (network functions). The term "reserved" indicates the interrupt is called by a certain system component (usually DOS), but the interrupt's use was never documented. In other words, we know who is using it, but we don't know why.

Summary Of Interrupts		
No.*	Address*	Purpose
00	000 - 003	Processor: Division by zero
01	004 - 007	Processor: Single step
02	008 - 00B	Processor: NMI (Error in RAM chip)
03	00C - 00F	Processor: Breakpoint reached
04	010 - 013	Processor: Numeric overflow
05	014 - 017	Hardcopy
06	018 - 01B	Unknown instruction (80286 only)
07	01D - 01F	Reserved
08	020 - 023	IRQ0: Timer (Call 18.2 times/sec.)
09	024 - 027	IRQ1: Keyboard
0A	028 - 02B	IRQ2: 2nd 8259 (AT only)
0B	02C - 02F	IRQ3: Serial port 2
0C	030 - 033	IRQ4: Serial port 1
0D	034 - 037	IRQ5: Hard drive

Summary Of Interrupts		
No.*	Address*	Purpose
0E	038 - 03B	IRQ6: Diskette
0F	03C - 03F	IRQ7: Printer
10	040 - 043	BIOS: Video functions
11	044 - 047	BIOS: Determine configuration
12	048 - 04B	BIOS: Determine RAM memory size
13	04C - 04F	BIOS: Diskette/hard drive functions
14	050 - 053	BIOS: Access to serial port
15	054 - 057	BIOS: Cassettes/extended function
16	058 - 05B	BIOS: Keyboard inquiry
17	05C - 05F	BIOS: Access to parallel printer
18	060 - 063	Call ROM BASIC
19	064 - 067	BIOS: Boot system (Ctrl+Alt+Del)
1A	068 - 06B	BIOS: Prompt time/date
1B	06C - 06F	Break key (not Ctrl-C) pressed
1C	070 - 073	Called after each INT 08
1D	074 - 077	Address of video parameter table
1E	078 - 07B	Address of diskette parameter table
1F	07C - 07F	Address of character bit pattern
20	080 - 083	DOS: Quit program
21	084 - 087	DOS: Call DOS function
22	088 - 08B	Address of DOS quit program routine
23	08C - 08F	Address of DOS Ctrl-Break routine
24	090 - 093	Address of DOS error routine
25	094 - 097	DOS: Read diskette/hard drive
26	098 - 09B	DOS: Write diskette/hard drive
27	09C - 09F	DOS: Quit program, stay resident
28	0A0 - 0A3	DOS: DOS is unoccupied
29-2E	0A4 - 0BB	DOS: Reserved
2F	0BC - 0BF	DOS: Multiplexer
30-32	0C0 - 0CB	DOS: Reserved
33	0CC - 0CF	Mouse driver functions
34-40	0D0 - 0FF	DOS: Reserved
41	104 - 107	Address of hard drive table 1
42-45	108 - 117	Reserved
46	118 - 11B	Address of hard drive table 2
47-49	11C - 127	Can be used by programs

Summary Of Interrupts		
No.*	Address*	Purpose
4A	128 - 12B	Alarm time reached (AT only)
4B-5B	12C - 16F	Free: can be used by programs
5C	170 - 173	NETBIOS functions
5D-66	174 - 19B	Free: can be used by programs
67	19C - 19F	EMS memory manager functions
68-6F	1A0 - 1BF	Free: can be used by programs
70	1C0 - 1C3	IRQ08: Realtime clock (AT only)
71	1C4 - 1C7	IRQ09: (AT only)
72	1C8 - 1CB	IRQ10: (AT only)
73	1CC - 1CF	IRQ11: (AT only)
74	1D0 - 1D3	IRQ12: (AT only)
75	1D4 - 1D7	IRQ13: 80287 NMI (AT only)
76	1D8 - 1DB	IRQ14: Hard drive (AT only)
77	1DC - 1DF	IRQ15: (AT only)
78-7F	1E0 - 1FF	Reserved
80-F0	200 - 3C3	Used within the BASIC interpreter
F1-FF	3C4 - 3CF	Reserved
*= All addresses and numbers in hexadecimal notation		

Hardware interrupts

Hardware interrupts are produced by various hardware components and passed, by the interrupt controller, to the processor. In this section we'll explain the steps involved in this process and the differences between PC/XTs and ATs.

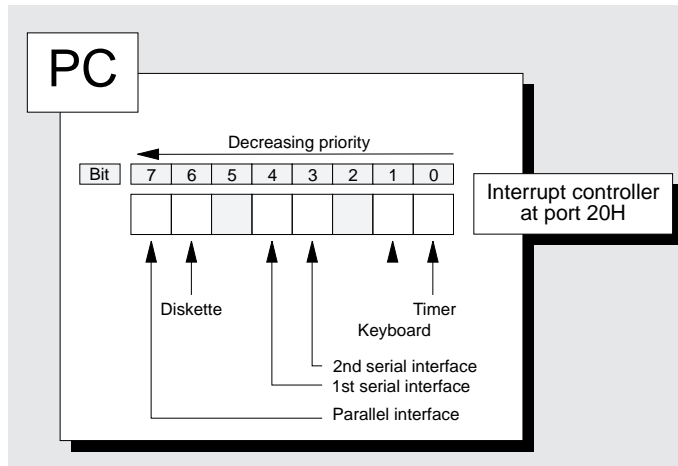
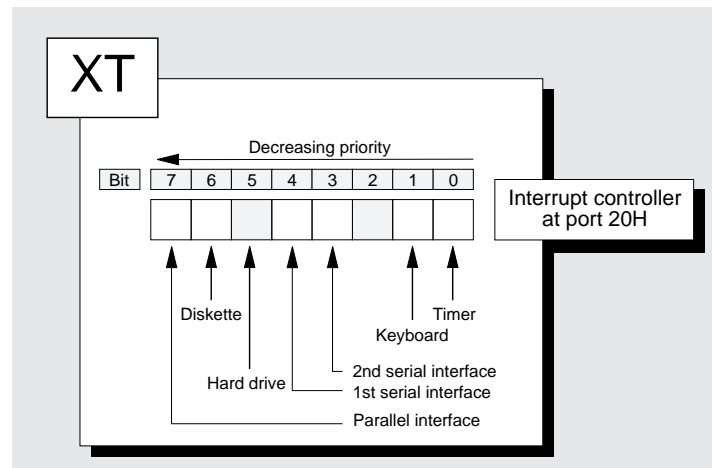
PC/XT hardware interrupts

Hardware interrupts 8 to 15 are called by the interrupt controller. Up to eight devices (interrupt sources) can be connected to the PC interrupt controller using interrupt lines IRQ0 to IRQ7. The device on line IRQ0 has the highest priority. The device connected with IRQ7 has the lowest priority. For example, if two interrupt requests arrive on lines IRQ3 and IRQ5, IRQ3 is addressed first. The number of the interrupt results from adding 8 to the IRQ number (in this case, it's interrupt 11).

Disabling hardware interrupts

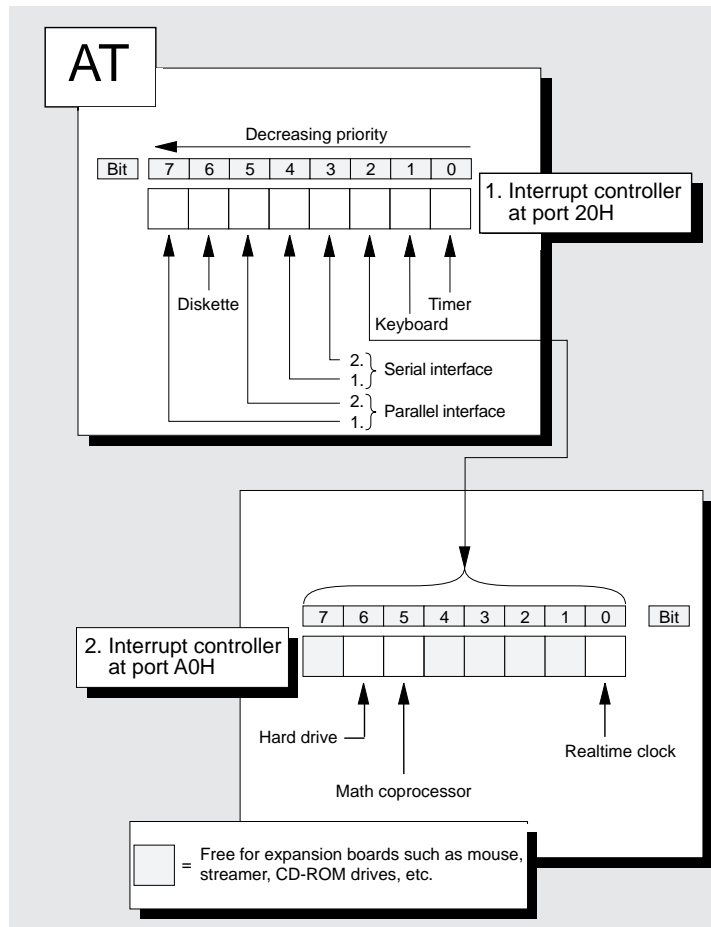
It's possible for a program to prevent the execution of hardware interrupts. This is useful when program execution shouldn't be interrupted. The processor will release a hardware interrupt, upon request from the interrupt controller, only if the interrupt flag is set in the processor's flag register. If the software has cleared the flag, the interrupt controller won't receive the requested interrupt.

You can also block single interrupts by programming the interrupt mask register in the interrupt controller.

PC interrupt requests and priorities*XT interrupt requests and priorities***AT hardware interrupts**

ATs have two 8259 interrupt controllers, which provide 16 interrupt sources. The eight additional interrupts are labeled IRQ08 to IRQ15. When an interrupt request addresses the second interrupt controller, it emulates an IRQ2 from the first interrupt controller. All the interrupt requests of the second interrupt controller are assigned a higher priority than lines IRQ4 to IRQ7 of the first interrupt controller.

If a request for IRQ2 is granted, the interrupt handler of interrupt 10 is executed. This interrupt handler first reads some of the registers of the second interrupt controller to determine the number of the IRQ. Based on the IRQ number, one of interrupts 70H to 77H is called as a software interrupt. It doesn't matter the call was actually initiated by the hardware because the device is waiting for execution of "its" interrupt handler. However, as a result of this procedure, the IRQ2 is unavailable to the first interrupt controller. So 15 interrupt sources are supported instead of 16.

AT interrupts and priorities

System Interaction

Now that we've discussed the essentials of system programming, let's see how DOS, BIOS, and the different levels of hardware communicate to give programs easy access to PC hardware. We'll use the keyboard as an example, since hardware interrupts, DOS, and BIOS functions are all involved. Let's follow the path of a character from the keyboard hardware to the program that reads the entered character and displays it on the screen.

Keyboard hardware

The keyboard hardware consists of the keyboard's processor. It's connected to the PC's processor by a cable. The keyboard processor monitors the keyboard and reports each key that is pressed or released to the system. The keyboard processor assigns a number instead of a character to each key. Control keys, such as **Ctrl** or **Shift**, are treated like any other key.

When the user presses a key, the keyboard processor passes the key number to the processor as a make code. (See Chapter 5 for more information on make codes.) When the user releases the key, the processor passes a break code. There is a minor difference between these codes. Although both use numbers between 0 and 127 for the key, the break code includes bit 7. To initiate the transfer, the keyboard controller first sends an interrupt signal to the interrupt controller, which arrives at line IRQ2. If hardware interrupts are enabled and a higher priority interrupt request doesn't exist, the processor then executes interrupt 09H.

BIOS keyboard handler

Interrupt 09H is a BIOS routine called the *keyboard handler*. The keyboard processor passes the key code to port 60H using the keyboard cable. It then calls the interrupt handler. From there, the BIOS handler reads the number of the key that was pressed or released. The rest of the system cannot use the key number because keyboards generate different numbers. So, the keyboard handler must convert the code into a character from the ASCII character set in a form the system can understand.

When you press a key, this key code is passed to the CPU as a byte. When you release the key, the processor passes the code to the CPU again, along with an added 128. This is the same as setting bit 7 in the byte. The keyboard instructs the 8259 interrupt controller the CPU should activate interrupt 9H. If the CPU responds, we reach the next level because a BIOS routine is controlled through interrupt 9H. While this routine is being called, the keyboard processor sends the key code to port 60H of the main circuit board using the asynchronous transmission protocol. The BIOS routine checks this port and obtains the number of the depressed or released key. This routine then generates an ASCII code from this key code.

This task is more complicated than it first appears because the BIOS routine must test for a control key, such as **Shift** or **Alt**. Depending on the key or combination of keys, either a normal ASCII code or an extended keyboard code may be required. The extended key codes include any keys that don't input characters (e.g., cursor keys).

Keyboard buffer

Once BIOS determines the correct code, this code is passed to the 16-byte BIOS keyboard buffer, which is located in the lower area of RAM. If it's full, the routine sounds a beep that informs the user of an overflow in the keyboard buffer. The processor returns to the other tasks that were in progress before the call to interrupt 09H.

BIOS keyboard interrupt

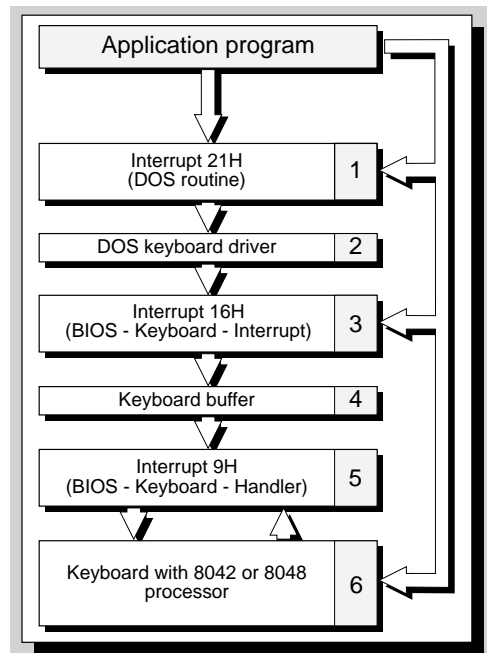
The next level, BIOS interrupt 16H, reads the character in the keyboard buffer and makes it available to a program. This interrupt includes three BIOS routines for reading characters, as well as the keyboard status (e.g., which control keys were pressed), from the keyboard buffer. These routines can be called with an INT assembly language instruction.

DOS level

The keyboard's device driver routines represent the DOS level. These DOS routines read a character from the keyboard and store the character in a buffer using the BIOS functions from interrupt 16H. In some cases, DOS routines may clear the BIOS keyboard buffer. If the system uses the extended keyboard driver ANSI.SYS, this keyboard driver can translate certain codes (e.g., function key **F1**) into other codes or strings. For example, it's possible to program the **F10** key to display the DIR command on the screen. Although, theoretically, you can call device driver functions from within a program, DOS functions usually address these functions.

DOS is the highest level you can go. You'll find the keyboard access functions in DOS interrupt 21H. These functions call the driver functions, transmit the results and perform other tasks. For example, characters and strings can be read and displayed directly on the screen until you press **Enter**. These strings are called by a program and complete a long process.

Keyboard access using the three-layer model



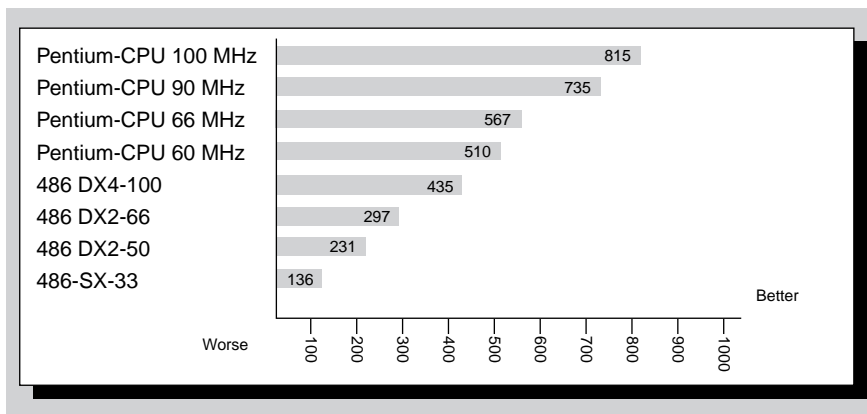
Pentium Processor

The technical possibilities of a PC have changed again with the Pentium processor. The Pentium was introduced by Intel in 1993. It features 100 MIPS (Million Instructions Per Second) at a clock speed of 66 MHz. This makes the Pentium almost twice as fast as a 486 DX2/66 in integer performance. The differences are even more significant in floating-point performance. Depending on the instruction mix, the Pentium beats its predecessor by three to seven times. Also, it's completely binary compatible with the 486, 386, 286, and even the 8086.

When asked about the performance of the Pentium, Intel has a very simple answer: 567. This measurement is a result of the ICOMP test developed by Intel for its own processors. This test, geared entirely to Intel's own processors, flows into the ICOMP index. As the following illustration shows, the measured value for the 66 MHz Pentium surpasses that of an equally fast 486 by almost double.

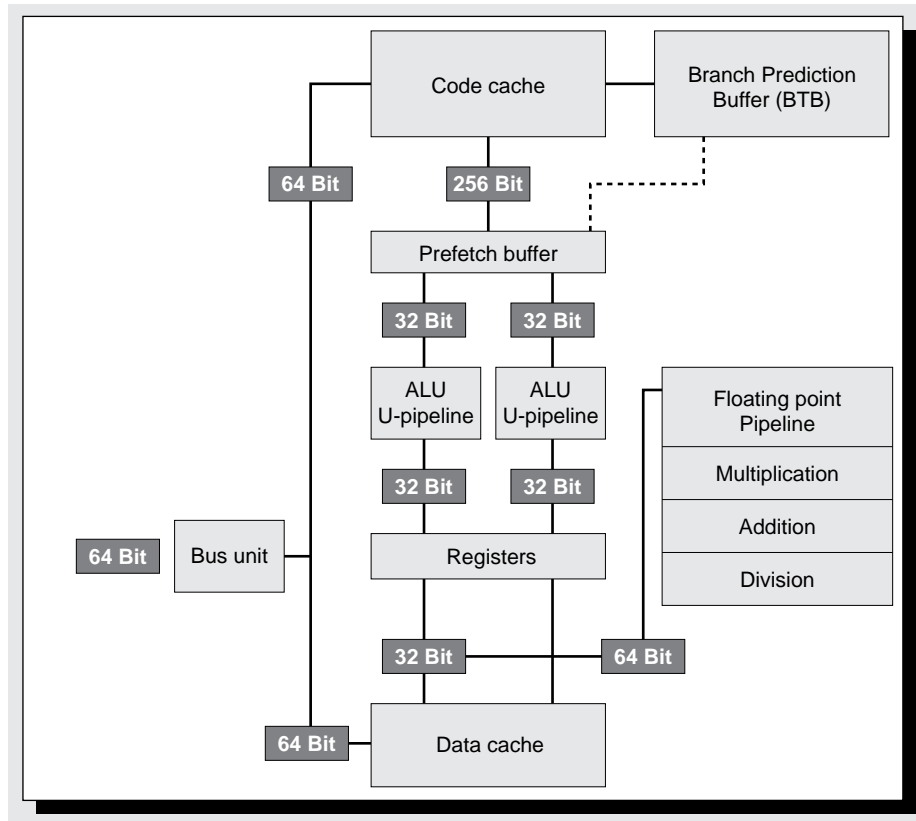
However, be careful when interpreting absolute data, such as the information returned by the ICOMP index. After all, selecting the processor test for such a benchmark is a subjective process, even if the manufacturers claim to be simulating real application conditions. Also remember, each manufacturer is eager to show its product is the best. So, they may downplay the performance areas in which its chip suffers compared to the competition or simply omit these performance areas.

*The Intel
ICOMP index*



On the whole, however, the direction in which this index is moving compared to the previous Intel processors might be correct, although you can't assume that doubling processor performance from the 486 to the Pentium could be duplicated at the user and software levels. There are numerous hardware and software components between the CPU and the user. These components either benefit only partially from the processor's performance, or they don't benefit at all. For example, this applies to all expansion boards. However, the Pentium has definitely advanced the PC world to previously unattainable dimensions. You're probably wondering what makes the Pentium so fast. Three components are responsible for the Pentium's speed: Superscalar integer execution unit, the first level processor cache, and the superscalar floating-point execution unit. We'll discuss these features in detail in this chapter.

Block diagram of the Pentium processor



First, let's review the most important facts about Intel's new "miracle chip":

- The Pentium is manufactured in 0.8-micron BiCMOS submicron technology. The traces or signal paths are only 0.8 millionths of a meter wide, or eight thousands of a millimeter wide.
- The processor is completely binary compatible with its predecessors in relation to instruction set, register, addressing modes, and operating modes.
- The processor still works with 32-bit registers and 32-bit addressing, but can be connected to a 64-bit data bus, enabling faster communication with memory.
- A superscalar architecture based on two parallel integer pipelines. In ideal circumstances, this would allow simultaneous execution of 2 machine language instructions in one cycle.
- The chip has a total of 3.1 million transistors.
- Two separate 8K data and code caches, in conjunction with the 64-bit bus interface (port), provide fast and continuous memory access.
- A special protocol called MESI (Modified, Exclusive, Shared, Invalid) ensures that a Pentium processor will work smoothly with other processors in a multiprocessor system.
- An improved floating-point unit executes commands significantly faster than the 486 and even provides the option of simultaneous execution of two instructions, although this happens on a limited scale.

Program execution

Program execution through the Pentium processor is based on a superscalar architecture with two parallel, five-stage integer pipelines that are connected with the processor cache and a branch target buffer (BTB).

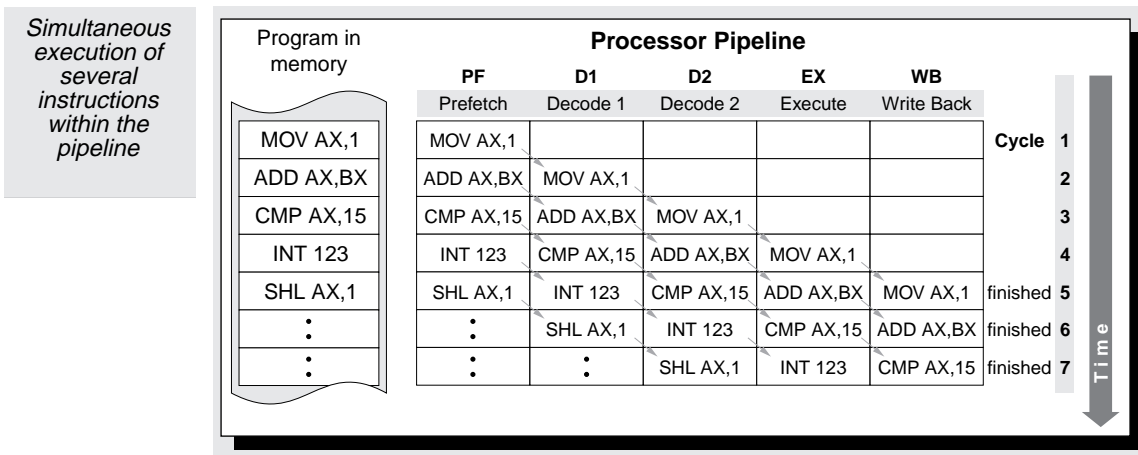
Execution in the pipeline procedure

To understand this efficient, expensive mechanism of program execution, you must first know how a microprocessor executes programs and machine language instructions. Although this process appears as a monolithic block from the outside, in the interior of the processor it is divided into five stages. The 486 and Pentium both have five stages that each instruction undergoes during its execution in a set sequence. These stages are abbreviated to PF, D1, D2, EX and WB. The following table shows the five stages of instruction execution on the 486 and the Pentium:

PF	Prefetch	D1	Decode1	D2	Decode2	EX	Execute	WB	Writeback
----	----------	----	---------	----	---------	----	---------	----	-----------

The execution of an instruction begins in the PF stage, the "instruction prefetch". In this stage, the machine language instruction is fetched from memory to the processor for execution. Once the instruction reaches the processor, it enters D1 stage, the first phase of instruction decoding. In this phase, the objective is to evaluate (analyze) the instruction, thus determining what kind of action it is supposed to trigger. Depending on the type of instruction, the next job is to determine the operands of the instruction (e.g., for a displacement memory address). This is the task of the second stage of instruction decoding, called D2. In the next pipeline stage, called EX, the execution of the instruction takes place, along with the associated memory accesses. In the WB stage, execution of the instruction concludes, with the contents of the processor register and the internal status register being updated.

The processor requires one cycle per stage to run these stages, while stages D2 and EX can also require one extra cycle, depending on the type of instruction. This provides a minimum of five cycles. However, if you check the Intel manuals, you'll discover that many instructions are executed in significantly fewer cycles. Some instructions even require only one or two cycles. Now we must determine how this is possible, if all the stages of the pipeline are necessary.



The solution is found in a principle used in assembly line production. Instead of only one instruction, as many instructions as the pipeline has stages runs through the various stages of the pipeline. So the subsequent instruction isn't processed after the preceding instruction leaves the last stage of the pipeline. Instead, it is processed immediately after the first stage of the pipeline. This means the different stages of the pipeline are busy at all times, always executing their function on a different instruction.

The instructions still require a minimum of five cycles to run through the complete pipeline, but because the pipeline finishes executing an instruction with each cycle, the instructions seem to require only one cycle for execution.

Superscalar pipelines

While the pipeline procedure of the 486 is already extremely fast, the Pentium multiplies this procedure by setting up a second, parallel pipeline. This is where the phrase "superscalar pipeline architecture" comes from. To keep the two pipelines separate, the first is called the "U pipeline" and the second one is called the "V pipeline."

With the help of these two pipelines, the Pentium should theoretically be able to execute two instructions simultaneously and, as a result, double the execution speed. However, in reality, this process isn't that easy. Frequently two sequential instructions can only be executed in sequence because they are dependent on each other. A simple example of this would be two machine language instructions, the first one describing a processor register on which the second instruction performs a read access. There are many other rules that make simultaneous execution of two sequential commands seem impossible. One such rule is the limitation of parallel execution to "simple" machine language instructions. Some examples of simple machine language instructions are MOV instructions, integer addition and subtraction, PUSH and POP instructions, and others. Only these instructions are actually "threaded" in the processor; all others are executed by Microcode, which is a type of processor operating system. It controls execution of complex machine language instructions through different execution units of the processor.

The second stage of the pipeline, D1, determines whether a parallel execution of both instructions is possible. In the PF stage, the current instruction to be executed and its successor are loaded into two parallel decoding units. This establishes the exact sequence. The current instruction goes to the decoding unit of the U pipeline and its successor goes to the decoding unit of the V pipeline.

If it is determined in D1 that simultaneous execution of the two instructions is possible, each of the two instructions then passes the various stages of its pipeline in parallel. If parallel execution is not possible, the instruction from the U pipeline goes to the next stage, while the instruction from the V pipeline is executed in the U pipeline as the instruction following the current instruction.

So the program code determines whether two instructions can be executed simultaneously or whether they have to pass the various stages of the U pipeline in sequence. Optimizing compilers for the Pentium consider this by organizing the machine code in such a way the sequential machine language instructions permit simultaneous execution as often as possible.

Branch Target Buffer

The efficiency of the pipeline principle is based upon the constant provision of new instructions to the pipeline. Only when the various stages of the pipeline are permanently filled does it seem possible the various instructions can be executed in one cycle. That is why two prefetch buffers are preset to the first stage of both pipelines. These prefetch buffers load the next instruction for the pipeline from memory or the processor-specific cache.

However, even these aren't helpful when the processor has to execute a jump instruction. In this case, instead of continuing with the following instruction, program execution continues with an entirely different instruction. As a result, execution of the following instructions, which are already in the pipeline, must be canceled and the pipeline must be loaded with new instructions. It takes a few cycles before the first instruction leaves the pipeline after the jump instruction.

Pentium uses a Branch Target Buffer (BTB) to avoid the problem of jump instructions. This buffer is used in the D1 stage of instruction execution for all types of NEAR jump instructions (i.e., for conditional and unconditional jumps, as well as for procedure references). If the processor encounters such an instruction in the D1 stage, it uses the address of the instruction in memory to search the BTB for the instruction. Every time the processor executes one of these jump instructions, it stores both the instruction's address and the jump destination's address in the BTB. If the instruction is registered there because it has already been executed, the processor assumes the jump should be executed again. Instead of loading the successor of the jump instruction into the pipeline, the processor loads the command to the target address.

However, if the jump instruction isn't registered in the BTB, the subsequent instruction is loaded in the pipeline. During the EX stage (at the latest), the processor will determine whether to execute the jump. If the processor predicted accurately with the address from the BTB, the instruction that follows the jump instruction will already be in the pipeline. So program execution can immediately continue. Even execution of a conditional jump will only take one cycle in this case.

However, if the processor's prediction is incorrect, this means the wrong commands are in the pipeline. So the pipeline must be "flushed." This involves canceling the execution of the commands currently in the pipeline and completely reloading the pipeline. As a result, instead of only one cycle, at least three cycles are needed to execute the jump command.

The Cache

The Pentium processor has two separate 8K caches: One for data and one for program code. Both of these caches are two-way set-associative caches. Each path consists of 128 entries with a cache line size of 32 bytes. The data cache can be operated in Writethrough and in Copyback mode, and is capable of responding to two accesses from the U and V pipeline of the processor simultaneously. To guarantee this, each cache line of 32 bytes is divided into eight 4-byte blocks.

If you're a computer expert, the previous explanation reveals the most important information about the cache structure of the Pentium processor. However, the explanation is extremely confusing to average computer users. You've probably never encountered the terms "cache lines", "two-way associativity", and "Copyback mode." Therefore, in the following sections we'll discuss how a processor cache operates and discuss Pentium cache in detail. This information may not improve your programming skills. However, if you want to know what makes the Pentium so fast, you must understand the cache.

Also, since the on-chip cache was first used with the 486, we'll also briefly discuss the processor cache of the i486.

Processor cache, hard drive cache, font cache, and CD-ROM cache are different devices that use the term "cache." A cache accelerates access to specific data and information by holding a portion of the data in a reserved section of memory. This process provides faster access than the actual storage device. This means that, for example, a hard drive cache reserves sectors of a hard drive, which have already been read in RAM memory, to deliver the sectors directly from this memory to the caller for a new read request instead of getting the sectors from the hard drive. Because a hard drive is several hundred times slower in access time than RAM, you can use this method to save a great deal of time.

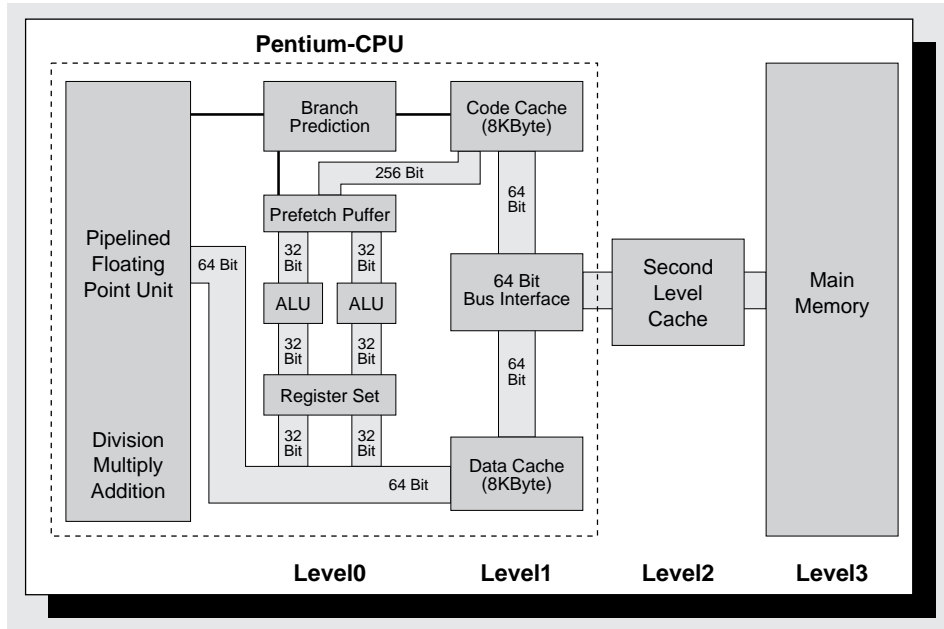
While hard drive, CD-ROM, and font caches use RAM memory as "high-speed memory", this doesn't apply to the processor cache. From the processor's view, it requires a cache because RAM doesn't supply data and program instructions fast enough for its purposes. This cache stores the memory locations the processor addressed during the last memory accesses. As a result, the next time the processor needs to access these memory locations, it doesn't have to get them from RAM. Instead, the processor can take the memory locations directly from high-speed cache memory.

However, not all processor caches are the same. It makes a big difference whether you are dealing with a primary or secondary processor cache. These are sometimes also called "first level cache" and "second level cache."

Currently 128K or 256K caches always refer to secondary cache. This is the cache that is between the processor and RAM and usually consists of SRAM (a form of high-speed RAM). The main memory is equipped with lower-priced DRAM chips, which are three to four times slower in supplying data to the processor than SRAM chips. This is where the speed advantage of a secondary cache becomes important. For comparison, consider that while SRAM is able to produce response times between 20 and 25 nanoseconds (millionths of a second), most PCs use 70ns, 80ns, or 100ns DRAM chips as main memory.

While secondary cache memory is located outside the CPU, the primary cache refers to the memory located directly on the CPU. The CPU can read from primary cache memory just as fast as from its registers. This is why it would be best to place the entire cache memory of a system directory on the processor, or better still, all the RAM memory. However, considering the current status of processor technology, this is impossible.

*First level,
second level
and third level
caches*



There is also a third level cache, which refers to normal main memory (RAM). This serves cache memory for hard drives and other peripherals. The numbering sequence is intentional, because the higher the number, the farther the cache is from the processor. As the number increases, the cache memory speed decreases, as does the price for 1K of the cache memory.

Cache effectiveness

The quality and effectiveness of a cache is measured from the ratio of cache hits and cache misses. A cache hit occurs when the data requested by the processor is already in a cache. So, the processor doesn't have to access slower memory. A cache miss means the data is not reserved in the cache, so first it must be loaded from memory into the cache, before it can be passed on to the processor. The greater the number of cache hits in comparison with cache misses, the more often the processor can be served from high-speed memory, ultimately causing it to work faster.

The ratio between cache hits and misses mainly depends on three factors: Organization of the cache, the type of program code being executed, and, obviously, the size of the cache. The third factor can be checked off quickly, because a growing cache size also increases the probability the information, for which the processor is searching, is already in the cache.

For the second factor, the type of program code, the "locality" of this code is very important. First, remember that a process cache not only caches the data that a program reads from memory during its execution, but also the executed program code. Regardless of whether the processor reads a variable or the next machine language instruction, they both must be furnished from memory. Also, in both cases, the cache first checks whether the address has already "been there" once.

This is why self-contained program sections, especially loops, that fit in the cache can be executed so quickly. If the execution of programs mainly occurs in blocks that aren't bigger than the cache, the existence of the cache will increase the speed of program execution. However, if a program continually jumps back and forth between different program sections, the cache won't be as noticeable.

There are two other factors that are basic prerequisites for the efficient use of cache memory. These two factors fall into the category of "Cache Organization". The first factor is cache strategy, in relation to read and write accesses, while the second factor is cache architecture, i.e., the way cached information is stored in the cache.

Cache strategies

Writethrough and WriteBack caches are related to the read and write accesses of the CPU. Writethrough is the simpler type, because the cache is addressed only for read accesses of the CPU. The cache transfers write accesses directly to main memory (RAM). Before doing this, however, the cache checks whether the specified memory location is already stored in the cache as a result of a read access. If this is the case, the new value of the memory location must also be entered in the cache.

If this doesn't happen, the cache contents and the contents of conventional memory may be inconsistent, which is the worst thing that can happen to a cache. Because of this inconsistency, the next time you read access the cache, it will return the old contents of the memory location, while conventional memory already contains a completely different value.

Along with the pure Writethrough procedure, Intel 80486 processors and above support a slightly modified procedure called "buffered writethrough." To speed up write accesses to memory, the first-level cache of the processor is equipped with additional write buffers. The 486 has four of these buffers. When data must be written to memory, the cache first places the data in one of these write buffers. This lets the CPU continue working immediately, because this memory can be addressed very quickly, similar to cache memory. While the CPU works, the cache writes the contents of the write buffer to conventional memory on its own, as soon as the bus is free. As long as this buffer doesn't fill up because the CPU is attempting to write data to memory faster than the data can be transported from the write buffer, the CPU's write operations to conventional memory won't be affected.

The Writeback procedure competes with the Writethrough procedure. For read operations, a Writeback cache acts just like a Writethrough cache. However, the two caches handle write operations differently. If the information to be written to conventional memory is already in the cache, it is first updated only in the cache. The information doesn't go to memory until the cache is forced to remove the memory location from cache memory because it needs space for new entries as a result of a read access by the CPU. If a memory location is written over and over again, this saves you the trouble of relatively slow write accesses to conventional memory until the time the memory location has to leave the cache. To keep this from taking too long, a type of write buffer called a castoff buffer is installed. The data are first stored in this buffer and then transferred to conventional memory in parallel with the work of the CPU.

Cache architecture

Cache memory is usually organized into cache lines; each line can receive information from conventional memory that is cached during a read or write operation. The size of a cache line depends on the internal data capacity of the CPU or the capacity of its primary cache. On the 80386 the cache lines are 32-bit (one DWORD = 4 bytes), on the 486 they are 128-bit (4 DWord = 16 bytes), and on the Pentium the cache lines are 256-bit (4 QWORD = 32 bytes).

For a read access to memory, the entire cache line is always filled, even when the processor requested only a single byte. Modern processors support "burst mode", which dramatically speeds up access to byte sequences in memory. Usually the CPU must place the address on the bus before reading out the desired memory location. However, in a burst access, the data are read as a block. The CPU only has to place the address for the first byte on the bus; the memory automatically furnishes all subsequent memory locations upon request.

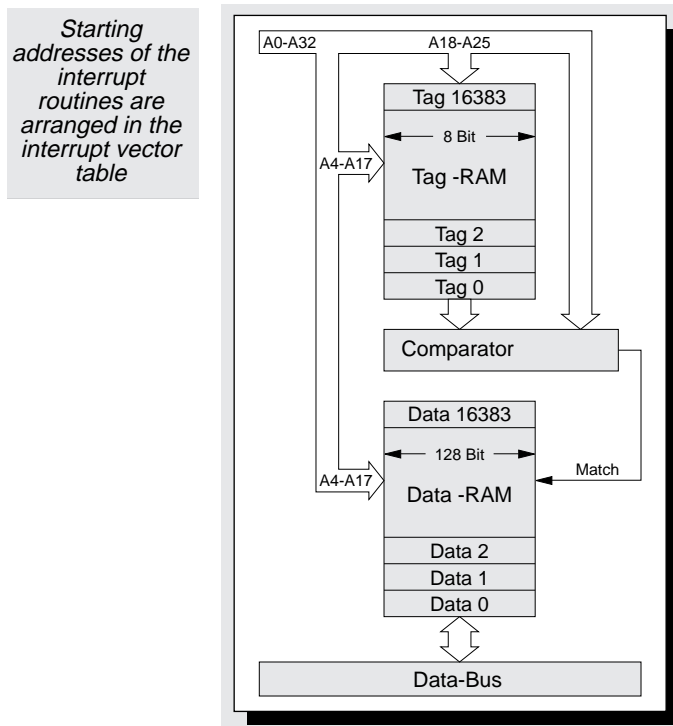
For example, the 486 usually requires 2 clock cycles to read a DWord from memory, so 4*2 clock cycles are necessary to fill a cache line. In burst mode, two cycles are required only for the first DWord; the three following words will be furnished in one cycle. That's why burst mode is also called a 2-1-1-1 burst; it takes only five cycles instead of the normal eight. The same procedure can also be used for write accesses.

Along with cached data, the cache must also store the memory addresses for the data. Each cache line is connected with a tag. This is where the cache stores the address of the data, as well as additional status information. (We'll talk about this information later in this chapter.) In secondary caches, the tags are not included with the cache lines. Instead, they are housed in separate memory components, which work even faster than the actual cache memory. In searching the cache for a memory location, the address not only has to be read out from the tag, but also must be compared with the address of the particular access by using a comparator. Naturally, this is time-consuming but is compensated for by speedier SRAM memory.

Along with cache lines and tags, a cache also always has a cache controller. A secondary cache usually has a microcontroller on the motherboard, while on a primary cache the controller is part of the processor. The controller controls communication with the CPU as well as the comings and goings of the cached information in the cache lines. It is the controller that translates the cache strategy into action and manages the pool of cache lines in accordance with a specific pattern.

Cache line organization

To determine the best possible cache line organization, first you must understand the cached information cannot be saved in any cache line you choose. Otherwise, in a read access the cache controller would be forced to run through all the tags in search of the correct address and compare the addresses stored there with the CPU address. This process would take more time than loading the information directly from conventional memory.



For this reason, cache controllers always connect the addresses of the cached memory locations with the cache lines, in which the addresses are stored. In the simplest type of cache organization, called "direct mapping", each byte from conventional memory has only one cache line in which it can be stored.

The cache controller checks this cache line when the CPU performs a read access. If the address is not listed there, it isn't in the cache.

In a direct mapped cache, mapping between the address and the cache line, in which it is stored, takes place via the memory address. The address is broken down into various components. To describe this process, we'll use a 256K secondary cache for a 486 system as an example.

Direct mapped secondary cache for the 486

Secondary caches for the 486 work with a cache line size of 128 bits (16 bytes). So, a 256K cache provides 16,384 cache lines. The cache controller's task is to clearly map the CPU address to one of the 16,384 cache lines. Since 16,384 is 2 to the 14th power, the lower 14 bits of the CPU address

determine the number of the cache line. However, instead of bits 0 to 13, these are bits 4 to 17. Bits 0 to 3 are needed to form the offset in the cache line; these four bits contain precisely the value between 0 and 15 that is needed for addressing the desired byte within the specific cache line.

Bits 0 to 3 make up the index in the cache line and bits 4 to 17 are used as an index in the cache line pool. So bits 18 to 31 remain. Actually, these bits are supposed to be stored in the tag of a cache line. However, instead of the 14 bits, frequently only eight bits (bits 18 to 25) are stored there. This means the cache can manage only the lower 64 Meg (2^{26}) of RAM memory, since there usually aren't even enough sockets provided on the motherboard for this much memory.

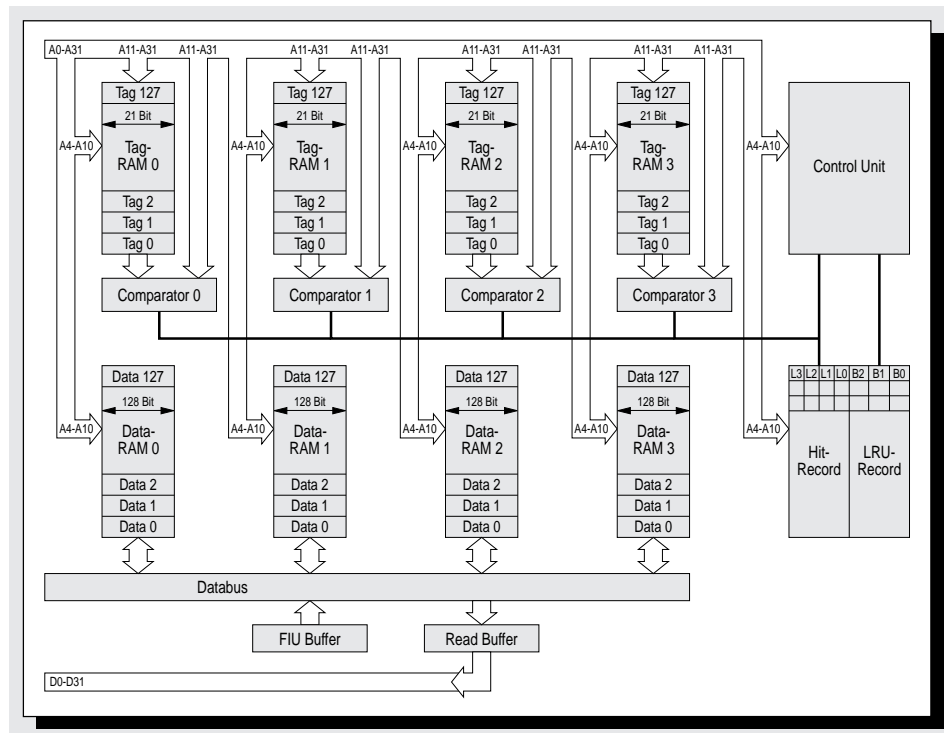
While its simplicity makes this procedure appealing, it does have a big disadvantage. Because the 64 Meg of RAM are mapped only to 16,384 cache lines, 256 memory locations share the same cache line. These memory locations are always 256K apart. However, once an address is loaded into the cache, whose cache line is already loaded with another one of these 256 addresses, it forces the old address out of the cache.

Associative caches

To prevent memory addresses from excluding each other in advance, associative cache memory refines the direct mapping process. Instead of assigning a single cache line to each memory location, it assigns each memory location two, four, or even eight possible cache lines. These are also called twofold, fourfold, or eightfold associative caches. An example of such a cache is the primary, fourfold associative cache of the 486, which holds 8K.

An associative cache requires much more circuitry than a direct mapped cache. In searching for a memory location, the cache controller must read two, four, or eight tags (depending on associativity), rather than one tag. Then the controller uses a comparator to compare them with the specific CPU address (or part of it).

The four-way associative 8K first level cache of the 486



In addition, for a read access, the cache controller must choose which of the potential cache lines it will place the memory location(s) in, since it can be assumed that all imaginable cache lines are already occupied. Instead of randomly forcing one of the filled cache lines out of the cache, most cache controllers implement an LRU (Last Recently Used) algorithm. LRU means the cache line which hasn't had a read hit for the longest time is removed from the cache. Next to the address in the tag, a couple of bits are also stored; the bits contain the sequence of the last accesses. On the whole, these measures significantly increase the effectiveness of an associative cache compared to a direct mapped cache.

Paging and interleaving

Paging or interleaving are other terms that are frequently used to describe the architecture of a cache. Both terms refer to the same concept, describing the distribution of the contents of various cache lines to different pages in memory. A page is a continuous block of memory; it's not the different cache lines that are divided, but their contents.

For example, the first level data cache of the Pentium is interleaved eightfold, meaning that eight DWORDs of a 256-bit cache line are also placed in eight separate pages. The first DWORDs from all cache lines are stored in the first page, the second DWORDs are stored in the second page, etc. This is done on the Pentium to enable simultaneous access to the cache from

the U and V pipeline of the processor. As long as the U and V pipeline want to read different DWORDs from one of the cache lines, they access different pages so they can both be operated at the same time.

MESI protocol

The cache's greatest difficulties are caused by external memory accesses that bypass the processor and cache controller. What is written to memory during such accesses could destroy the consistency of the cache (i.e., the information stored in the cache would no longer match the actual contents of RAM). DMA controllers can cause such inconsistencies by bypassing the processor to write data to memory from an external device, such as a hard drive controller. However, bus masters on bus systems, such as EISA and MCA, can also destroy the consistency of a cache. In the bus mastering design, the CPU briefly passes bus control to the bus master. Usually the bus master is a component of an add-in board and it uses the control over the bus to shift data within RAM as quickly as possible, or to transfer data from its own memory to RAM.

To eliminate inconsistencies resulting from such accesses in advance, the cache controllers of secondary caches are linked to the system in such a way they handle DMA transfers and bus master accesses. However, in multiprocessor systems, which will become more important in the era of the Pentium and Windows NT, this is not possible, because the CPUs are directly on the bus. Therefore, they cannot be connected to the bus from the cache controller.

Also, with multiprocessor systems, each processor has its own first level cache and consistency between these different caches (and RAM) must be preserved. INTEL solves this problem with the Pentium processor by using Pentium:MESI protocol, which the Pentium supports for synchronization of caches in a multiprocessor system. MESI protocol has a feature called bus snooping, which is a procedure that helps a processor and other system components prompt for and manipulate the status of cached information in the caches of other processors.

We'll use the following example to illustrate this:

Two Pentium processors running in parallel cache a specific memory location simultaneously. One of the two processors modifies this memory location. Since the cache is operating in write-back mode, the memory location doesn't get updated in RAM until later. This makes the memory location in the cache of the second processor invalid, since it still has the old value. If the processor doesn't realize this, it will inevitably result in a conflict if the processor continues processing this memory location.

However, when the different processors communicate with each other by using MESI protocol, such inconsistencies are avoided. The acronym MESI stands for the four different states that a cache line of the processor cache can have, M, E, S and I. Each cache line has a tag containing the appropriate flags for identifying this state. The following is an explanation of each letter:

M - Modified

The cache line is only in this cache, but it has been modified and not yet written back to RAM. As a result, the contents in RAM no longer match the current contents of the memory location.

E - Exclusive

The cache line is exclusively in this one cache and hasn't been modified. The contents of RAM and the contents of the cache line still match.

S - Shared

The cache line may still be in other caches and hasn't been modified. A write access must take place in write-through mode (i.e., must be passed on directly to RAM). All other caches containing this cache line will recognize the change and automatically update the contents of their cache lines.

I - Invalid

The contents of the cache line are invalid; it is empty and free to receive new data.

First level cache of the Pentium

Now that we've discussed the principle and structure of first and second level caches, you may better understand the information presented at the beginning of this chapter. Now we'll discuss how a first level cache is implemented in the Pentium. Actually the Pentium has two separate first level caches: One for data and one for code. Both caches are 8K and two-way associative. Each path contains 128 cache lines of 32 bytes each ($2 \text{ paths} * 128 \text{ cache lines} * 32 \text{ bits} = 8\text{K}$).

Both caches can be prompted at the same time, while the data cache is capable of responding simultaneously to two requests from the U and V pipeline of the processors. To achieve this purpose, its cache lines are eightfold interleave, permitting simultaneous access to each DWord in the cache. The tags in the data cache are even triple-ported, which means they can be addressed by three sources at the same time. Two of these sources are the U pipeline and the V pipeline, while the third source is used for bus snooping when it is necessary to determine whether a specific address is in the cache.

You can switch each cache line in the data cache to Writethrough or Writeback mode using software or hardware. While operating the cache in Writeback mode makes sense from a performance standpoint, it can lead to problems with specific memory areas. For example, consider the video RAM on a graphics card. If this memory area is cached in Writeback mode, the cached information takes quite a while to get to video RAM, which, in turn, slows down composition of the screen.

Overall, the cache architecture in the Pentium is much more complicated than that of its predecessor, the 486. The double integer pipeline and the concept of using the Pentium in multiprocessing systems contribute to this factor. Nevertheless, the cache is an important driving force behind the outstanding performance of the Pentium.

Floating-point unit

The floating-point unit of the Pentium is integrated on the chip, just like the 486. However, the performance of this unit has been significantly improved compared to its predecessor. The following table demonstrates this by showing a comparison of execution times for floating-point instructions on a 486 and on a Pentium. Intel claims the execution of floating-point instructions on the Pentium is up to seven times faster than the 486, enabling the Pentium to compete with workstation processors.

The following table compares execution times for floating point instructions on a 486 and a Pentium. The FXCH command has a special position, since it is frequently used in floating-point programming. The reason for this has to do with the organization of the eight floating-point registers for all Intel processors and numerical coprocessors. These registers are handled like a stack; most floating-point instructions use the top of the stack as one of their arguments and also place their result there. As a result, a program must always take values to the top of the floating-point stack or move the values from there. Because the FXCH instruction handles this task, it is executed more frequently than all other floating-point instructions.

Command	486	Pentium	Command	486	Pentium
FXCH	4	1	FSUB	10	3
FLD	3	1	FMUL	16	3
FST	3	2	FDIV	73	39
FADD	10	3			

That is also why developers increased the speed of executing this instruction on the Pentium significantly over the 486's execution speed. The Pentium requires only one cycle, and sometimes doesn't even need any cycle at all. What makes this possible is the floating-point unit's ability to run the FXCH command parallel to another floating-point instruction. However, this is also the only case in which both floating-point pipelines can be occupied simultaneously with the execution of two floating-point instructions.

The superscalar, eight-stage floating-point pipeline forms the foundation for parallel execution of an FXCH instruction and any other floating-point instruction. Like an integer pipeline, the floating-point pipeline consists of two pipelines working in parallel. Actually, the floating-point pipeline shares its first five stages with the integer pipeline, but also requires three additional stages to complete execution of a floating-point instruction.

The following table shows the eight stages of the floating-point pipeline. The first three stages are identical to the execution of an integer command, because this is when the CPU finds out that it is dealing with a floating-point instruction. In the fourth pipeline stage (EX), in which the integer commands are executed, depending on the command, the floating-point unit first fetches the operands of the floating-point instruction from memory or a register and converts them into a special floating-point format, with which the floating-point operates internally. The actual execution of the instruction takes place in stages X1 and X2. In the WF stage, the result of the floating-point operation is then rounded off and transferred to the target register on the floating-point stack. Execution of the floating-point is completed in the ER stage, in which any errors that may have occurred in the operation are reported and the floating-point status register is updated.

PF	Prefetch	X1	Floating-point Execution Stage 1
D1	Decode1	X2	Floating-point Execution Stage 2
D2	Decode2	WF	Write File
EX	Execute	ER	Error Report

Other features

In addition to its superscalar architecture, first level cache, and floating-point unit, the Pentium has several other features that distinguish it from its predecessors. They are:

- Paging in Protected and Virtual 86 mode is no longer limited to 4K pages, but can also operate with a page size of 2 Meg or 4 Meg. This should help reduce the management time necessary for paging in multitasking systems.
- The Pentium has a system management mode, as already implemented in special versions of the 486. It helps integrate a Pentium processor in programs designed to save power.
- The "Function Redundancy Check" allows parallel operation of two Pentium processors that check up on each other to ensure correct operation. This should spur the development of error-tolerant systems.
- Improvements in debugging support searching for complex errors and debugging with hardware add-ons.
- In Performance monitoring, the Pentium measures the progress of program execution.

2

System Programming In Practice

Now that you know some basics, we can look at the practical side of system programming: Program development in BASIC, Pascal and C. Each language has its own commands, procedures and functions for addressing memory, reading ports or calling interrupts.

QuickBASIC

QuickBASIC isn't the best language to use for system programming because it's more limited than Pascal or C. However, system programming in BASIC is possible even if you cannot do everything that you can in Pascal or C. For example, BASIC doesn't have direct pointer access. In this book, you'll find fewer demonstration programs in BASIC than in Pascal and C. We included any programs that could be translated into BASIC.

The BASIC demonstration programs we list and include on the companion CD-ROM run under the QuickBASIC interpreter Version 4.5. However, these programs don't run under Microsoft's QBasic interpreter (QBasic isn't able to call interrupts). Most of these programs require that you run the QuickBASIC environment while loading a library named QB.LIB:

```
QB /L QB
```

QuickBASIC data types

When you call interrupt functions, you must be familiar with the processor data types. Interrupt functions are written in assembly language and no other data types are available at that level of programming. So, if you want to perform system programming in QuickBASIC, you must copy the QuickBASIC data types to the data types of the processor. The table on the right shows which types correspond.

QuickBASIC Type	Stored as
String * 1	BYTE
Integer	WORD
Long	DWORD

Unlike Pascal and C (the char type), QuickBASIC doesn't recognize single characters. The String * 1 type compensates for this limitation. String * 1 is a string the length of a byte. The QuickBASIC compiler views this string in memory as a single byte.

However, it's more difficult to operate one of these strings than a normal byte. The reason for this is that a numeric value can only be loaded into a variable declared in this way using the CHR\$() function, as the following example shows:

```
DIM byte AS STRING * 1
byte = CHR$(5)      'This is O.K.                - Program runs if you enter this
byte = 5             'Error: Type mismatch        - Program does not run
```

You can derive the value of such a byte only by using the ASC function:

```
DIM byte AS STRING * 1
byte = CHR$(13)
IF ASC(byte) = 13 THEN PRINT 13  'This is O.K.                - Program runs if you
                                '                                enter this
IF byte = 13 THEN PRINT 13      'Error: Type mismatch - Program does not run
```

Working with the integer and long data types is also difficult if you use them to reproduce words and dwords. QuickBASIC views the highest bit as the type of number (positive or negative) and views the number as negative when that bit is set. For example, if you receive a word after the interrupt call and bit 15 is set in this word (indicating that the value is greater than 32,768), QuickBASIC views the number as negative. The same problem occurs with dwords, only less frequently. (A number with bit 31 set is much larger than you'd normally see in system programming.)

You can manage integer types by converting them into floating point numbers with a function. Check the sign bit, make the conversion, and continue processing. The following MakeWord function listed appears under different names, such as GetWord, in some of the BASIC demonstration programs listed on the companion CD-ROM:

```
FUNCTION MakeWord& (ANum AS INTEGER)

  IF ANum < 0 THEN
    MakeWord = 65536& + ANum
  ELSE
    MakeWord = ANum
  END IF
END FUNCTION
```

You pass the integer, which may have a set bit 15, to the function. The function returns a positive long data type because the function assumes that bit 31 specifies the sign.

Strings

Most DOS and BIOS functions expect strings as a sequence of bytes containing the ASCII codes of the individual characters and terminated by a null byte (a byte consisting of the value 0). System programming books often call this type of string an ASCIIZ string (ASCII-Zero) string.

BASIC stores strings in a different format, from which you must distinguish between variable length strings and fixed length strings. System programming always uses fixed length strings because it's easier to calculate their addresses in memory than variable length strings. You need the string addresses to pass them to a DOS or BIOS function (more on this later). If you declare a string of fixed length in your programs, QuickBASIC reserves that many bytes of the string. The following reserves 20 bytes, whose contents are undefined:

```
DIM S AS STRING * 20
```

Adding the following loads the contents of the array S in these 20 bytes, padding the remainder of the allocated string with spaces:

```
S = "PC Intern"
```

Adding the null byte to create an ASCIIZ string requires special handling. We need a WHILE+WEND loop to locate the last character of the string, then we must add the null byte to the string using the MID\$ statement:

```
DIM S AS STRING * 20
DIM I AS INTEGER

S = ""
INPUT "Please enter a string"; s

I = LEN(S) - 1
WHILE (I > 0) AND (MID$(S, I, 1) = " ")
  I = I - 1
WEND
IF I = 0 THEN I = 1
MID$(S, I) = CHR$(0)
```

However, if you know the string and don't want the user to enter it, you can include the null byte in the string allocation:

```
S = "PC Intern" + CHR$(0)
```

Structures and arrays

Similar to applications and other programs, DOS and the BIOS manage much information using structures and arrays. The following table shows an example of a structure returned to the caller by DOS. This information occurs when you browse through directories looking for files.

Directory entry structure as returned by DOS functions 4EH and 4FH		
Address	Contents	Type
00H	Reserved	21 bytes
15H	Attribute byte of the file	1 byte
16H	Time of last modification	1 word
18H	Date of last modification	1 word
1AH	File size	1 dword
1EH	Filename and extension separated by a period but without a path specification (ends with a null byte)	13 bytes
Length: 43 bytes		

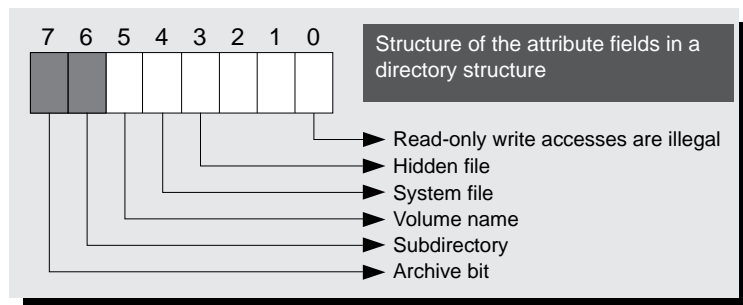
The following program listing excerpt shows how this structure can be recreated in QuickBASIC (you'll find this structure in the DIRB.BAS program discussed later in this book):

```
TYPE DirStruct
  Reserved AS STRING * 21
  Attrib AS STRING * 1
  Time AS INTEGER
  Date AS INTEGER
  Size AS LONG
  DatName AS STRING * 13
END TYPE
```

As you can see, the Reserved element at the beginning of the DirStruct structure is represented by a fixed length string. This is the easiest way to reserve a specific number of bytes. The rest of the elements in the DirStruct structure refer to the various components of the DOS structure in their data types. Bytes are reproduced as String * 1, words as INTEGERS, and dwords as LONGs. The names of the individual fields are unimportant. You can choose any name you want because the names don't affect the structure. Obtaining a correct reproduction of the structure is important.

Accessing bit fields

In structures, fields often represent bit fields, in which individual bits or groups of bits have a specific meaning. The attribute byte in the previous directory structure also represents a bit field. As the illustration on the left shows, each single bit represents a certain file attribute. For example, a bit might provide information about whether the file is write/protected, is a system file, or even is a file (a subdirectory). You must know how to read the individual bits.



If you want to read a certain bit, first you must know its value. You know that bit 0 has a value of 1, bit 1 a value of 2, bit 2 a value of 4 and so on, until you reach bit 7, which has a value of 128. To determine whether you're dealing with a subdirectory, you must use the value of bit 4, which is 16.

You want to set all the attribute byte's other bits to 0. From there you can then determine whether bit 4 is set. The AND operator masks all bits not in the AND mask. The following expression unsets all bits except bit 4 (bit 4 = 16):

```
AttributeByte AND 16
```

If bit 4 is set, a result of 16 is returned. Otherwise, the result is 0.

We can apply this expression using IF+THEN+ELSE:

```
IF ( ( AttributeByte AND 16 ) <> 0 ) THEN
    'If the result <> 0 it's a subdirectory
ELSE
    'If the result = 0 there's no subdirectory
ENDIF
```

Unfortunately, checking more than one bit at a time complicates this process. The values of the different bits must be added together. For example, suppose that you want to determine whether the file is both hidden and a system file. The corresponding flags are stored in bits 1 and 2, and have a value of 6 when added together. The following expression returns the contents of both bits:

```
AttributeByte AND 6
```

This time, however, the expression used in the previous example cannot be directly applied to this example:

```
( AttributeByte AND 6 ) <> 0
```

This expression is already TRUE if one of the two bits is set and the result of the AND operation doesn't equal 0. However, if you want to know whether both flags were set, you must modify the process to something similar to the following:

```
IF ( ( AttributeByte AND 6 ) = 6 ) THEN
    'Hidden and System
ELSE
    'Not Hidden, not System
ENDIF
```

Often you'll want to set bits to pass a bit field to a DOS or BIOS function. Again, the main focus is on the values of the bits, but the OR operator performs this task instead of the AND operator. The following statement sets bit 3 of the attribute byte:

```
AttributeByte = AttributeByte OR 8
```

Again, to set multiple bits, the values must be added:

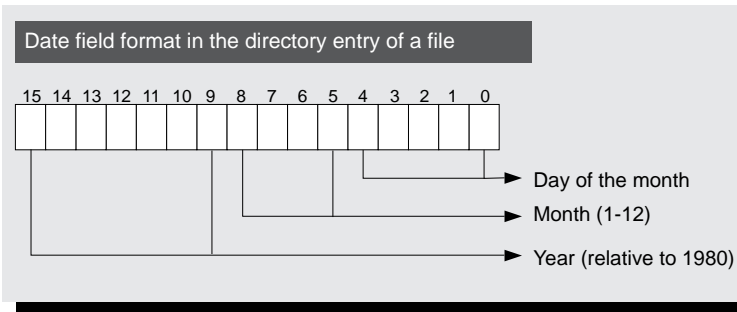
```
AttributeByte = AttributeByte OR ( 8 + 16 )
```

Both of these expressions set the desired bit to 1. Suppose that you want to set a bit to 0. To do this, use an AND operation in a different arrangement to mask the bit you want set to 0. According to the laws of binary logic, you must then invert the value using the NOT operator to achieve the desired result. To set bit 5 to 0, use the following statement:

```
AttributeByte = AttributeByte AND NOT(32)
```

Once again, you can mask more than one bit at a time using the following:

```
AttributeByte = AttributeByte AND NOT( 32 + 8 );
```



However, bit fields don't always consist of separate bits. Often they are comprised of bit groups, whose individual bits form a certain value when added together. An example of this is the date field in the directory entry of a file. This field contains three bit groups that specify the day, month, and year the particular file was created or last modified. So, to analyze this information, you must determine the value the three bit groups represent instead of checking the status of given bits.

You can easily determine the day by using the described procedure with an AND operator:

```
Day = DateField AND ( 1 + 2 + 4 + 8 )
```

When you want to determine the month, the AND operation is no longer sufficient because the isolated bit group must also be shifted to the right by five bits to obtain the number of the month. In BASIC, the only way to do this is by dividing the value by 2 raised exponentially by the number of bit positions (by which the value is to be shifted to the right). You can determine the month and the year by using the following statements:

```
Month = ( DateField AND ( 32 + 64 + 128 + 256 ) ) \ 32 '2^5 power = 32
Year = ( DateField AND ( 512+1024+2048+4096+8192+16384+32768 ) ) \ 512
```

However, you'll encounter problems again, at least with the second statement, because of the sign bit. This is why you should use the MakeWord function described earlier in this section:

```
Year = ( MakeWord(DateField) AND 65024& ) \ 512
```

To shift bits to the left instead of to the right, use multiplication instead of division. For example, the following will make a DateField out of a given day, month, and year:

```
DateField = Day + ( Month * 32 ) + ( Year * 512 )
```

Calling interrupts from QuickBASIC

The QuickBASIC QB.QLB quick library provides the INTERRUPT and INTERRUPTX statements for calling software interrupts. You can call all 256 Intel processor interrupts with these statements. To access this library, you must start QuickBASIC with the /L QB switch.

INTERRUPT and INTERRUPTX can also access interrupt 21H, which lets you call the DOS API (DOS Application Program Interface) functions. There are more than 200 of these functions, which refer to functions provided by DOS applications. The QB.BI include file lets you access DOS API functions. You must include this file in your programs by using the following:

```
REM $INCLUDE: 'QB.BI'
```

The syntax for both statements is:

```
CALL INTERRUPT(Interruptnum, InReg, OutReg)
CALL INTERRUPTX(Interruptnum, InReg, OutReg)
```

Accessing the processor registers

The InReg and OutReg parameters used by the INTERRUPT statement are of type RegType, which represents a structure defined within QB.BI. The RegType structure makes the various processor registers available to a BASIC program. From

the InReg structure, the INTERRUPT command loads the processor registers with the specified values from the InReg structure. After the interrupt call, the OutReg structure contains the contents of the processor registers.

From the definition of RegType you may conclude that the different variables within this structure reflect the processor registers of the same name.

```
TYPE RegType
    ax AS INTEGER
    bx AS INTEGER
    cx AS INTEGER
    dx AS INTEGER
    bp AS INTEGER
    si AS INTEGER
    di AS INTEGER
    flags AS INTEGER
END TYPE
```

RegType accesses only the 16-bit registers instead of the 8-bit registers. So, to access an 8-bit register, you must use a 16-bit register. For example, the following lines load the value 1BH (&h1B) into the AH register by multiplying that register's value by 256, and then moving the value eight bit positions to the left:

```
DIM Regs AS RegType
Regs.AX = &h1B * 256
CALL INTERRUPT( &hxyz, Regs, Regs )    'The result of the interrupt call
```

However, this also sets AL to 0. If you don't want this to happen, set the desired value with OR:

```
DIM Regs AS RegType
Regs.AX = Regs.AX OR ( &h1B * 256 )
CALL INTERRUPT( &hxyz, Regs, Regs )    'The result of the interrupt call
```

Write the desired value to the AX register (and ensure that the AH register is empty) to access the AL register. However, if there is already a value in AH, you should use the OR operator to avoid destroying the contents of AH.

```
DIM Regs AS RegType
Regs.AX = &h1B                                'Load AL with 1BH (&h1B), assume AH = 0
Regs.AX = Regs.AX OR &h1B                      'AH remains unchanged
CALL INTERRUPT( &hxyz, Regs, Regs )    'The result of the interrupt call
```

You could also use the same principle with all the other general registers. For example, it's just as easy to determine the contents of the various 8-bit registers after an interrupt call. If you're interested in the high byte, simply divide the contents of the 16-bit register by 256. If you're interested in the low byte, you can mask the high byte with an AND operator.

```
DIM Regs AS RegType
CALL INTERRUPT( &hxyz, Regs, Regs )    'Interrupt call (replace &hxyz with
                                         'the interrupt of your choice

PRINT "AH = "; MakeWord(Regs.AX) \ 256
PRINT "AL = "; Regs.AX AND &HFF
```

Including the segment register

Maybe you've already noticed that the segment register is ignored in RegType. Numerous DOS and BIOS functions expect parameters in the DS and ES segment registers or return information to these registers. Because of this, there is a command called INTERRUPTX, which works exactly like INTERRUPT except that it works with structures of the RegTypeX type. Although RegTypeX is similar in structure to RegType, it also contains two fields for the ES and DS registers.

```

TYPE RegTypeX
    ax AS INTEGER
    bx AS INTEGER
    cx AS INTEGER
    dx AS INTEGER
    bp AS INTEGER
    si AS INTEGER
    di AS INTEGER
    flags AS INTEGER
    ds AS INTEGER
    es AS INTEGER
END TYPE

```

Reading the flags in the flag register

In many cases, the flag registers can also return information to the calling program. DOS functions extensively use the carry flag, which is set after the function is called and when the function call fails.

You can also access the various processor flags after calling INTERRUPT or INTERRUPTX using the FLAGS variable in the OutReg structure. You can calculate the contents of each flag using an AND operator with the value of the flag as the table on the right shows.

So, a test for the carry flag could look like the following:

```

DIM Regs AS RegType
CALL INTERRUPT( &hxyz, Regs, Regs ) 'The interrupt call
IF ( Regs.Flags AND 1 ) <> 0 THEN PRINT "Error"

```

Flag	Bit Pos.	Value
Carry	0	1
Parity	2	4
Auxiliary	4	16
Zero	6	64
Sign	7	128
Overflow	11	2048

Buffers and QuickBASIC

Many functions expect pointers to buffers when they are called. The functions either take information from the buffers or place information in the buffers (e.g., file contents). These pointers are always FAR; they consist of a segment address and an offset address. This FAR pointer data can be anywhere in memory, not necessarily in the current program's memory segment.

Passing pointers to interrupt functions

DOS function 09H is an example of a function that takes pointers. This function displays a string on the screen beginning at the current cursor position. Like all DOS functions, it expects the function number in the AH register and the address of the buffer containing the string to be displayed in the DS:DX register pair. DS takes the segment address of the buffer, and DX takes the offset address.

Although creating a string is easy in QuickBASIC, you may also want to know how to pass the buffer address. QuickBASIC provides the VARSEG and VARPTR functions, which supply the segment and offset addresses of the specified variable. The following program demonstrates how to use these functions with DOS function 09H (&H09) as an example. Unlike other DOS functions, function 09H looks for a \$ character, instead of a null byte, at the end of the buffer.

```

' 9HDEMOB.BAS

'$INCLUDE: 'QB.BI' 'Include file for interrupt call

DIM S AS STRING * 20
DIM RegsX AS RegTypeX

```

```
CLS
S = "PC Intern" + "$"
RegsX.AX = &H900           'Function number 09H
RegsX.DS = VARSEG(S)       'Segment address
RegsX.DX = VARPTR(S)       'Offset address
CALL InterruptX(&H21, RegsX, RegsX)
```

Receiving pointers from interrupt functions

The following program calls DOS function 1BH, which returns a pointer in the DS:BX register pair. This pointer points to a byte containing the media code of the current drive. DOS uses the media code to describe the different types of drives, with codes between F0H and FFH. The value F8H (248) characterizes all types of hard drives.

Since QuickBASIC doesn't recognize FAR pointers, use the PEEK() command to read the media ID. Although this command can be used to read the contents of any memory location, it accepts only one offset address and always accesses the "current" segment. Fortunately, you can define this segment using the DEF SEG command as the following program shows.

```
'MEDIAIDB.BAS

'$INCLUDE: 'QB.BI'           'Include file for interrupt call

DIM RegsX AS RegTypeX
DIM MediaID AS INTEGER

CLS
RegsX.AX = &H1B00           'Function number
CALL interruptx(&H21, RegsX, RegsX)
DEF SEG = RegsX.DS          'Define segment
MediaID = PEEK(RegsX.BX)    'Read media ID
PRINT "Media ID = "; MediaID
```

Turbo Pascal

Our discussion of Pascal is based on Borland's Turbo Pascal. Although Turbo Pascal compatible Pascal compilers are available, we'll concentrate on Turbo Pascal. All the Pascal programs we describe were developed using Turbo Pascal Version 5.5 but they will also run under the later versions of Turbo Pascal or Borland Pascal. Since the demonstration programs illustrate system programming, we omitted all the OOP (Object Oriented Programming) enhancements that are available in Turbo. Once you understand the logic of each demonstration program, you can add any extras, such as OOP objects, to suit your own needs.

Turbo Pascal data types

Similar to other compilers, Turbo Pascal's data types mostly correspond with processor data types, which allow fast and easy processing. The table at the top of the following page shows how Turbo Pascal stores the different data types. In Turbo Pascal, pointers are always FAR, regardless of whether they point at data or are procedural pointers, which refer to program code.

Strings

Most DOS and BIOS functions expect strings as a sequence of bytes, containing the ASCII codes of the individual characters, and terminated by a null byte (a byte consisting of the value 0). In system programming, this type of string is called an ASCIIZ (ASCII-Zero) string. Pascal also saves a string as a sequence of bytes, with each byte representing the ASCII codes of the characters. However, unlike ASCIIZ strings, the first byte indicates the string's length instead of the string null byte at the end of the string. Although this method is more practical for processing strings, it's incompatible with DOS and BIOS functions.

The following program listing shows how Pascal strings can be easily converted into ASCIIZ strings by simply adding a null byte. However, when passing such a string to a DOS or BIOS function using its address, you must specify the address of the first character (string[1]), instead of the address of the length byte (string[0]). We'll discuss this in detail after the program listing.

```
{*  ASZDEMO.PAS  *}

program ASZDemo;

var ASCIIZ : string[100];
    i : integer;

begin
  write ( 'String: ' );
  readln( ASCIIZ );
  ASCIIZ := ASCIIZ + chr(0);
  for i := 0 to ord( ASCIIZ[0] ) do
    begin
      write( i:2, '    ', ord( ASCIIZ[i] ):3 );
      if ( ASCIIZ[i] > ' ' ) then
        write( ' ', ASCIIZ[i] );
      writeln;
    end;
  end.
```

Pascal Type	Stored as
CHAR	BYTE
BYTE	BYTE
BOOLEAN	BYTE
INTEGER	WORD
WORD	WORD
LONGINT	DWORD
POINTER	DWORD

The following shows the screen output that's created by the previous program after it's compiled and called. The program prompted the user for a string. After the user typed the string and pressed **Enter**, the program added a null byte and displayed the string on the screen, including the length byte and null byte.

*Output of the
ASZDEMO
program*

```
String: ASCIIZ string    <---- Prompt and input
0  14                    <---- Length byte
1  65 A
2  83 S
3  67 C
4  73 I
5  73 I
6  90 Z
7  32
8  115 s
9  116 t
10 114 r
11 105 i
12 110 n
13 103 g
14  0                    <---- Added null byte
```

Structures and arrays

Similar to applications and other programs, DOS and the BIOS manage much information using structures and arrays. The most important factor lies in the compiler's creating the information in the sequence specified, aligning each field on a word boundary. Although Turbo Pascal has a compiler directive for aligning data (the {A\$} directive), this directive usually doesn't work on structures and arrays.

The following table shows an example of a structure returned to the caller by DOS.

Directory entry structure as returned by DOS functions 4EH and 4FH		
Address	Contents	Type
00H	Reserved	21 bytes
15H	Attribute byte of the file	1 byte
16H	Time of last modification	1 word
18H	Date of last modification	1 word
1AH	File size	1 dword
1EH	Filename and extension separated by a period but without a path specification (ends with a null byte)	13 bytes
Length: 43 bytes		

The following program listing excerpt shows how this structure can be recreated in Pascal (you'll find this structure in the DIRP1.PAS program which we'll discuss later):

```

type DirBufType = record      { Data structures of functions 4EH and 4FH }
    Reserved : array [1..21] of char;
    Attr      : byte;
    Time      : integer;
    Date      : integer;
    Size      : longint;
    Name      : array [1..13] of char
end;

```

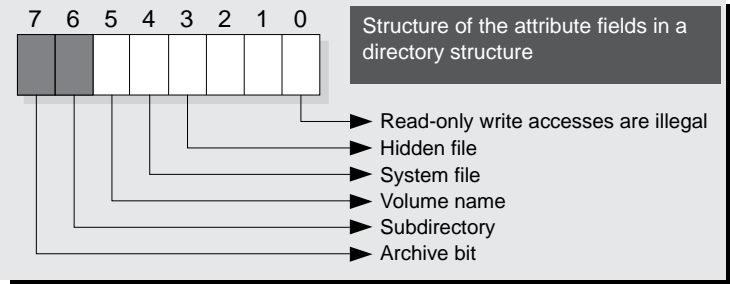
As you can see, the Reserved element at the beginning of the DOS structure is represented by an array. This can be either a char or byte array. The fields within this structure must have the same offset address, which means that these elements are the same distance from the beginning of the structure as in the DOS structure. The rest of the elements in the DirBufType structure refer to the various components of the DOS structure in their data types. Bytes are reproduced as bytes, words as integers, and dwords as longints. The individual field names are unimportant. You can choose any name you want because the names don't affect the structure. Obtaining a correct reproduction of the structure is what matters.

Accessing bit fields

Fields often represent bit fields in structures. Individual bits or groups of bits have a specific meaning in these structures. The attribute byte in the previous directory structure also represents a bit field. As the following illustration shows, each single bit represents a certain file attribute. For example, a bit may provide information about whether the file is write/protected, is a system file, or even is a file (a subdirectory). You must know how to read the individual bits.

If you want to read a certain bit, first you must know its value. You know that bit 0 has a value of 1, bit 1 a value of 2, bit 2 a value of 4, and so on until you reach bit 7, which has a value of 128. To determine whether you're dealing with a subdirectory, you must use the value of bit 4, which is 16.

You want to set all the attribute byte's other bits to 0. From there you can then determine whether bit 4 is set. The AND operator masks all bits not in the AND mask. The following expression unsets all bits except bit 4 (bit 4 = 16):



```
AttributeByte and 16
```

If bit 4 is set, a result of 16 is returned. Otherwise, the result is 0.

This expression can be used as follows within an if loop:

```
If AttributeByte and 16 <> 0 then
  { If the result <> 0 it's a subdirectory }
else
  { If the result = 0 there's no subdirectory }
```

Checking more than one bit at a time complicates this process. The values of the different bits must be added together. For example, suppose that you want to determine whether the file is both hidden and a system file. The corresponding flags are stored in bits 1 and 2, and have a value of 6 when added together. The following expression returns the contents of both bits:

```
AttributeByte and 6
```

This time, however, the expression used in the previous example cannot be directly applied to this example:

```
AttributeByte and 6 <> 0
```

This expression is already TRUE if one of the two bits is set and the result of the AND operation doesn't equal 0. However, if you want to know whether both flags were set, you must modify the process to something similar to the following:

```
If AttributeByte and 6 = 6 then
  { Hidden and System }
else
  { not Hidden, not System }
```

Often you'll want to set bits to pass a bit field to a DOS or BIOS function. Again, the main focus is on the values of the bits, but the OR operator performs this task instead of the AND operator. The following statement sets bit 3 of the attribute byte:

```
AttributeByte := AttributeByte or 8;
```

Again, to set multiple bits, the values must be added:

```
AttributeByte := AttributeByte or ( 8 + 16 );
```

Both expressions set the desired bit to 1. Suppose that you want to set a bit to 0. Use an AND operation in a different arrangement to mask the bit you want set to 0. According to the laws of binary logic, you must then invert the value using the NOT operator to achieve the desired result. To set bit 5 to 0, use the following statement:

```
AttributeByte := AttributeByte and not( 32 );
```

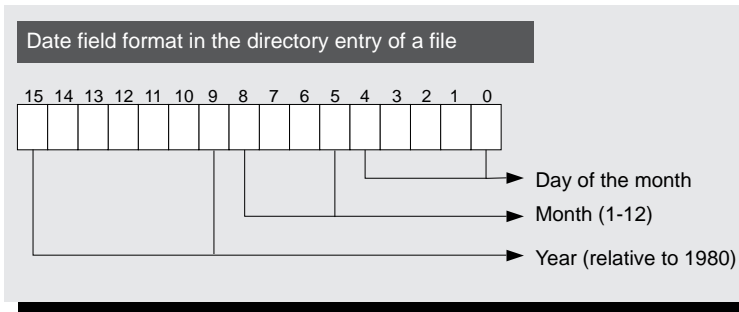
Once again, you can mask more than one bit at a time using the following:

```
AttributeByte := AttributeByte and not( 32 + 8 );
```

However, bit fields don't always consist of separate bits. Often they are comprised of bit groups, whose individual bits form a certain value when added together. An example of this is the date field in the directory entry of a file. This field contains three bit groups that specify the day, month, and the year the particular file was created or last modified. So, to analyze this information, you must determine the value the three bit groups represent instead of checking the status of given bits.

You can easily determine the day by using the described procedure with an AND operator:

```
Day := DateField and ( 1 + 2 + 4 + 8 );
```



When you want to determine the month, the AND operation is no longer sufficient because the isolated bit group must also be shifted to the right by five bits to obtain the number of the month.

The SHR operator in Turbo Pascal shifts an expression to the right by any number of bits.

Determine the month and the year by using the following statements:

```
Month := ( DateField and ( 32 + 64 + 128 + 256 ) ) shr 5;
Year := ( DateField and ( 512+1024+2048+4096+8192+16384+32768 ) shr 9;
```

The SHL operator, which is the opposite of the SHR operator, shifts a value to the left bit by bit. For example, you can use this operator to create a date field from a given day, month, and year:

```
DateField := Day + ( Month shl 5 ) + ( Year shl 9 );
```

Calling interrupts from Turbo Pascal

Turbo Pascal provides the `Intr` and `MsDos` procedures, which are defined in the `DOS` unit. This unit also contains some type and constant declarations that are needed for calling types and constants.

The syntax for `Intr` is as follows:

```
Intr(InterruptNumber : byte, Regs : Registers);
```

The `InterruptNumber` parameter specifies the number of the interrupt to be called. Since every value between 0 and 255 is accepted for this parameter, you can call all available interrupts, including hardware interrupts. The `MsDos` procedure is a special form of the `Intr` procedure. You can call it the same way you call `Intr`:

```
MsDos(Regs : Registers);
```

Notice that unlike `Intr`, `MsDos` doesn't have an `InterruptNumber` parameter. `MsDos` accesses interrupt 21H, which lets you call the DOS API (DOS Application Program Interface) functions. There are over 200 of these functions, which refer to functions provided by DOS applications.

Accessing the processor registers

As you may conclude from the definition of `RegType`, the different variables within this structure reflect the processor registers of the same name. Both procedures expect a variable of type `Registers`, which is defined in the `DOS` unit. `Registers` accepts the values loaded in the processor registers before the interrupt call. Then these values are supposed to be passed to the called interrupt. After returning from `MsDos` or `Intr`, these variables contain the values that were in the various processor registers after the called interrupt function ends.

To simplify register addressing, `Registers` provides a variant record, in which the registers are listed with their normal names. `Registers` is defined as follows in the `DOS` unit:

```
type Registers = record
    case integer of
        0 : (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : word);
        1 : (AL, AH, BL, BH, CL, CH, DL, DH : byte);
    end;
```

The 16-bit processor registers AX to ES are represented by the word variables of the same name. The 8-bit processor registers AL to DH are represented by variables of type byte.

The divisions of 8-bit and 16-bit registers into half registers results in overlapping between both register groups in memory (i.e., two 8-bit variables overlap the corresponding 16-bit variable). So, AL and AH share the same memory space as AX. BL and BH share the same memory space of BX. This also applies to the CL/CH and DL/DH variables.

Notice the order in which 8-bit registers are specified. This order must mirror the format in which the 16-bit register is placed in memory above them. Since, in memory, the low byte of a word precedes the high byte, the L register must be declared before the corresponding H register.

If Regs is a variable of type Registers, you can easily address the single processor registers by the different components of this variable:

```
Regs.ax,      Regs.bx,      Regs.cx,
Regs.ah,      Regs.dl      etc.
```

To pass the value D3H to the DL register during an interrupt call, do the following:

```
Regs.DL := $D3;
```

Before calling an interrupt using Intr or MsDos, load the registers, which are used by the function you'll call, with the information you want passed to the function. The interrupt ignores all registers except those on which it directly relies.

Reading the flags in the flag register

In many cases, the flag registers can also return information to the calling program. DOS functions extensively use the carry flag, which is set after the function is called and when the function call fails.

To simplify checking the flags, the DOS unit defines different constants, which reflect the bit values of the processor flags: Use an AND operator to check whether one of these bits is set. The following expression sets the Boolean variable to TRUE when the carry flag is set.

```
Error := ( ( Regs.Flags and FCarry ) <> 0 );
```

Constant	Bit Pos.	Bit Value
FCarry	0	1
FParity	2	4
FAuxiliary	4	16
FZero	6	64
FSign	7	128
FOverflow	11	2048

Buffers and Turbo Pascal

Many functions expect pointers to buffers when they're called. The functions either take information from or place information in the buffers (e.g., file contents). These pointers are always FAR; they consist of a segment address and an offset address. This FAR pointer data can be anywhere in memory, not necessarily in the current program's memory segment.

Passing pointers to interrupt functions

DOS function 09H is an example of a function that takes pointers. This function displays a string on the screen, beginning at the current cursor position. Like all DOS functions, it expects the function number in the AH register and the address of the buffer containing the string to be displayed in the DS:DX register pair. DS takes the segment address of the buffer and DX takes the offset address.

Although creating a string in Pascal is easy, you may also want to know how to pass the buffer address. Turbo Pascal provides the Seg() and Of() functions, which supply the segment and offset addresses of any memory object. It doesn't matter whether you're working with a local or global variable or a typed constant.

The following program demonstrates how to use these functions with DOS function 09H (\$09) as an example. The program uses DOS function 09H to display the string from the Message variable on the screen. Unlike other DOS functions, function 09H looks for a \$ character, instead of a null byte, at the end of the buffer.

```
'9HDEMOP.PAS

program 9HDemoP;

uses DOS;

var Regs      : Registers;
    Message   : string[20];

begin
    Message := 'DOSPrint' + '$';

    Regs.AH := $09;
    Regs.DS := seg( Message[1] );
    Regs.DX := ofs( Message[1] );
    MsDos( regs );
end.
```

Receiving pointers from interrupt functions

The following program calls DOS function 1BH, which returns a pointer in the DS:BX register pair. This pointer points to a byte containing the media code of the current drive. DOS uses the media code to describe the different types of drives, with codes between F0H and FFH. The value F8H (248) characterizes all types of hard drives.

To determine the media ID from the returned pointer, the MediaPtr type is defined as a pointer to a byte at the beginning of the program. Since pointers are always FAR in Turbo Pascal, you can be certain that you've created a FAR pointer. The program defines MP as a variable of this type. After calling the DOS \$1B function, the program loads MP with the returned pointer from the register pair DS:BX. The program uses Turbo Pascal's Ptr function to do this. This function receives a segment and offset address and forms a generic pointer from them.

This pointer can be used to access the referenced information as in any normal pointer operation. The Writeln statement at the end of the program demonstrates this.

```
{*  MEDIAIDP.PAS      *}

program MediaIdP;

uses Dos;                                     { Add Dos unit }

type MediaPtr = ^byte;                       { Create a byte pointer }

var Regs : Registers;                        { Processor registers for interrupt call }
    MP   : MediaPtr;                         { Variable for media pointer }

begin
    Regs.AH := $1B;                          { Pass 1BH to AH register }
    MsDos( Regs );                          { Call DOS interrupt 1BH }
    MP := ptr( Regs.DS, Regs.BX );           { Read pointer }
    writeln( 'Media ID = ', MP^ );          { Display media ID }
end.
```

Accessing memory with Mem, MemW, and MemL

Turbo Pascal has three predefined arrays, called Mem, MemW, and MemL, that are used to access bytes, words, and longints (dwords). A special syntax is used to access these arrays within brackets and the segment address is separated from the offset address by a colon.

You could have accessed the media ID using the following in the MEDIAIDP.PAS program:

```
mem[ Regs.DX : Regs.BX ];
```

When accessing multiple pointers, Mem, MemW, and MemL will need a more complex syntax than the one previously shown.

Port access in Turbo Pascal

Turbo Pascal recognizes PC ports as a predefined array. However, Turbo Pascal also supports two arrays for port access: Port (for 8-bit ports) and PortW (for 16-bit ports). PortW allows you to send 16-bit values to ports, while Port only accepts 8-bit values. The array you select will depend on the expansion board or support chip you want to access. If the board or chip is 16-bits, you can use PortW; otherwise, you must use Port for access.

You can read information from ports and write information to ports using normal array syntax. For example, both of the following statements dump the contents of port 3C4H, which is part of the graphics controller on an EGA/VGA card:

```
XByte := port[ $3C4 ];
XWord := portw[ $3C4 ];
```

The following statements allow you to send a byte or word just as easily:

```
port[ $3C4 ] := XByte;
portw[ $3C4 ] := XWord;
```

Examples of these statements are located in Chapter 4.

C Language

Unlike Pascal, the market for C compilers is characterized by the rivalry between Microsoft and Borland. Both companies have several products on the market: Microsoft QuickC and Microsoft C 6.0, Borland Turbo C++ and Borland C++. Both C++ compilers preserve the compatibility with the standard (Turbo C) implementation.

The C programs in this book can be compiled under all the compilers we just named, although some warning messages may appear on the screen. All the programs were compiled under Microsoft C 6.00, with warning levels changed as needed. We also test-compiled programs using Borland's Turbo C++ and the default settings of the Turbo C++ environment.

These programs are affected by the differences between the libraries found in the Microsoft and Borland compilers. Because of this, some programs contain constructs like the following, which is taken from the DIRC2.C demonstration program listed later in this book. The differences between the two libraries are intercepted by defining macros.

```
#ifdef __TURBOC__
#define DIRSTRUCT
#define FINDFIRST( path, buf, attr )
#define FINDNEXT( buf )
#define NAME
#define ATTRIBUTE
#define TIME
#define DATE
#define SIZE
/* Turbo C Compiler? */
struct ffbk
findfirst( path, buf, attr )
findnext( buf )
ff_name
ff_attrib
ff_ftime
ff_fdate
ff_fsize
```

```

#else                                     /* No --> Microsoft C */
#define DIRSTRUCT                         struct find_t
#define FINDFIRST( path, buf, attr )      _dos_findfirst(path, attr, buf)
#define FINDNEXT( buf )                  _dos_findnext( buf )
#define NAME                             name
#define ATTRIBUTE                         attrib
#define TIME                             wr_time
#define DATE                             wr_date
#define SIZE                             size
#endif

```

Since the demonstration programs in this book illustrate system programming, we omitted all the OOP (Object Oriented Programming) enhancements available in Turbo C++. Once you see the logic of each demonstration program, you can add any extras, such as OOP objects, to suit your own needs.

C data types

Like all compilers, the C data types mainly correspond with processor data types, which allows fast and easy processing. The table on the right shows how various C compilers store the different data types. Since C doesn't have a byte type or word type, you'll find typedef functions, similar to the following, at the beginning of many of the C programs we use:

```

typedef unsigned char BYTE;
typedef unsigned int WORD;

```

These lines define the two types that are very important to system programming. In C, the memory model that's used governs the use of NEAR and FAR pointers. The programs in this book were developed using the SMALL memory model. So they work exclusively with NEAR pointers. When FAR pointers are needed for system programming, the far modifier is used in the variable declaration:

```

int far *p;          /* P is a FAR pointer */

```

We found that Microsoft QuickC doesn't like working with FAR pointers while the Options/Compiler Flags/Pointer Check option is enabled. Disable this option to avoid problems while executing the demonstration programs from this book.

Strings

Most DOS and BIOS functions expect strings as a sequence of bytes, containing the ASCII codes of the individual characters, and terminated by a null byte (a byte consisting of the value 0). In system programming, this type of string is called an ASCIIZ (ASCII-Zero) string. Since C stores strings in ASCIIZ format, they don't have to be converted.

Structures and arrays

Similar to applications and other programs, DOS and the BIOS manage much information using structures and arrays. The most important factor lies in the compiler's creating the information in the sequence specified, aligning each field on a word boundary.

All C compilers are familiar with compiler directives that can influence this structure. Microsoft C compilers support the /Zp directive, which ensures that the fields within structures aren't separated. Borland compilers have an option called Word alignment in the Options/Compiler.../Code generation... dialog box within the integrated development environment. Ensure that this option is disabled; otherwise the compiler will separate the fields.

The following table shows an example of a structure returned to the caller by DOS:

C Type	Stored as
unsignedchar	BYTE
char	BYTE
int	WORD
unsigned int	WORD
near *void	WORD
long	DWORD
far *void	DWORD

Directory entry structure as returned by DOS functions 4EH and 4FH		
Address	Contents	Type
00H	Reserved	21 bytes
15H	Attribute byte of the file	1 byte
16H	Time of last modification	1 word
18H	Date of last modification	1 word
1AH	File size	1 dword
1EH	Filename and extension separated by a period but without a path specification (ends with a null byte)	13 bytes
Length: 43 bytes		

The following excerpt from a program listing (DIRC1.C) which we'll describe later demonstrates how this structure can be reproduced in C.

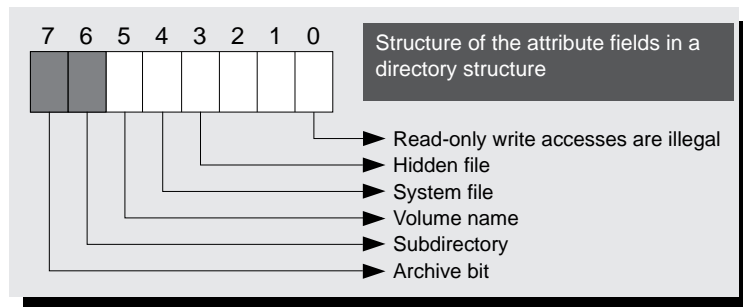
```
typedef unsigned char BYTE; /* Create a byte */
typedef struct { /* DIR structure for functions 4EH and 4FH */
    BYTE Reserved[21];
    BYTE Attribute;
    unsigned int Time;
    unsigned int Date;
    unsigned long Size;
    char Name[13];
} DIRSTRUCT;
```

As you can see, the Reserved element at the beginning of the DOS structure is represented by an array. You can use either a CHAR array or BYTE array. The various fields within the C structure must have the same offset addresses so they are the same distance from the beginning of the structure as in the DOS structure.

The remaining elements in the DIRSTRUCT structure correspond to the various components of the DOS structure in reference to their data types. Bytes are reproduced as bytes, words as unsigned ints, and dwords as unsigned longs. The names of the individual fields are unimportant. Since the names don't affect the structure, you can choose any name you want. Obtaining a correct reproduction of the structure is all that matters.

Accessing bit fields

In structures, fields often represent bit fields, in which individual bits or groups of bits have a specific meaning. The attribute byte in the previous directory structure also represents a bit field. As the illustration on the right shows, each single bit represents a certain file attribute. For example, a bit may provide information about whether the file is write/protected, is a system file, or even is a file (a subdirectory). You must know how to read the individual bits.



If you want to read a specific bit, first you must know its value. You know that bit 0 has a value of 1, bit 1 a value of 2, bit 2 a value of 4, etc., until you reach bit 7, which has a value of 128. To determine whether you're working with a subdirectory, you must use the value of bit 4, which is 16.

You want to set all the attribute byte's other bits to 0. Then you can then determine whether bit 4 is set. The AND operator (the & character in C) masks all bits that aren't in the AND mask. The following expression unsets all bits except bit 4 (bit 4 = 16):

```
AttributeByte & 16
```

If bit 4 is set, a result of 16 is returned. Otherwise, the result is 0. This expression can be used within an if loop as follows:

```
if ( ( AttributeByte & 16 ) != 0 )
    /* If the result <>0 it's a subdirectory */
else
    /* If the result = 0 there's no subdirectory */
```

Checking more than one bit at a time complicates this process. The values of the different bits must be added together. For example, suppose that you want to determine whether the file is both hidden and a system file. The corresponding flags are stored in bits 1 and 2, and have a value of 6 when added together. The following expression returns the contents of both bits:

```
AttributeByte & 6
```

This time, however, the expression used in the previous example cannot be directly applied to this example:

```
( AttributeByte & 6 ) != 0
```

This expression is already TRUE if one of the two bits is set and the result of the AND operation doesn't equal 0. However, if you want to know whether both flags were set, you must modify the process to something similar to the following:

```
If ( ( AttributeByte & 6 ) == 6 )
    /* Hidden and System */
else
    /* Not Hidden, Not System */
```

Frequently you'll want to set bits to pass a bit field to a DOS or BIOS function. Again, the main focus is on the values of the bits, but the OR operator (the | character in C), instead of the AND operator, performs this task. The following statement sets bit 3 of the attribute byte:

```
AttributeByte = AttributeByte | 8;
AttributeByte |= 8; /* Abbreviated version */
```

Again, to set multiple bits, the values must be added:

```
AttributeByte = AttributeByte | ( 8 + 16 );
```

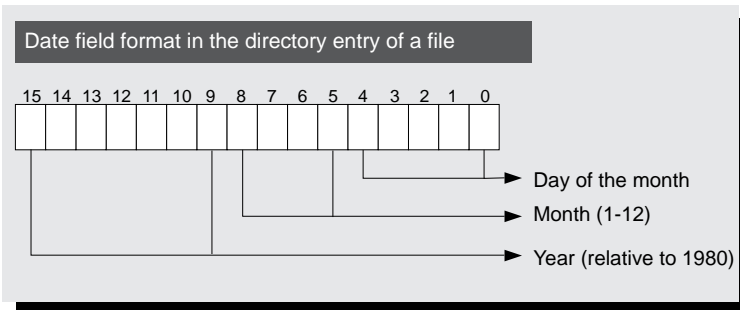
Both of these expressions set the desired bit to 1. Suppose you want to set a bit to 0. Use an AND operation in a different arrangement to mask the bit you want set to 0. According to the laws of binary logic, you must then invert the value using the NOT operator (the ! character in C) to achieve the desired result. To set bit 5 to 0, use the following statement:

```
AttributeByte = AttributeByte & !32;
```

Once again, you can mask more than one bit at a time using the following statement:

```
AttributeByte = AttributeByte & !( 32 + 8 );
```

However, bit fields don't always consist of separate bits. Often they are comprised of bit groups, whose individual bits form a certain value when added together. An example of this is the date field in the directory entry of a file. This field contains three bit groups that specify the day, month, and year the particular file was created or last modified. So, to analyze this information, you must determine the value the three bits represent instead of checking the status of the given bits.



You can easily determine the day by using the described procedure with an AND operator:

```
Day = DateField & ( 1 + 2 + 4 + 8 );
```

However, when determining the month, the AND operation is no longer sufficient because the isolated bit group must also be shifted to the right by five bits to get the number of the month. In C, the `>>` operator shifts an expression to the right by any number of bits. You can determine the month and the year by using the following statements:

```
Month = ( DateField & ( 32 + 64 + 128 + 256 ) ) >> 5;
Year = ( DateField & ( 512+1024+2048+4096+8192+16384+32768 ) ) >> 9;
```

The `<<` operator, which is the opposite of the `>>` operator, shifts a value to the left bit by bit. For example, you can use this operator to create a date field from a given day, month, and year:

```
DateField = Day + ( Month << 5 ) + ( Year << 9 );
```

Calling interrupts from C

Both Borland and Microsoft compilers provide the `int86()`, `int86x()`, `intdos()`, and `intdosx()` functions for calling software interrupts. While the `int86()` and `int86x()` functions can call all 256 interrupts of the Intel processor, the `intdos()` and `intdosx()` functions direct their attention to interrupt 21H (0x21), which lets you call the DOS API (DOS Application Program Interface) functions. There are over 200 of these functions, which refer to functions provided by DOS applications.

The declarations of these functions are in the `DOS.H` include files of both compilers, which must be linked to a C program to work with these functions. These declarations are as follows:

```
int intdos(union REGS *inregs, union REGS *outregs);
int intdosx(union REGS *inregs, union REGS *outregs, struct SREGS *sreg);
int int86(int, union REGS *inregs, union REGS *outregs);
int int86x(int, union REGS *inregs, union REGS *outregs, struct SREGS *sreg);
```

Accessing processor registers

All four procedures expect pointers to structures of type `REGS`, while the two functions that end with "x" expect a variable of type `SREGS`. These are structures that reproduce the processor registers.

From the first passed structure (`inregs`), the functions load the various processor registers before the interrupt call, while they load the contents of the processor registers in the second passed structure (`outregs`) after the call.

To make it easier to address both the 8-bit and the 16-bit registers, `REGS` represents a union in which two structures of `WORDREGS` and `BYTEREGS` type can be placed on top of each other:

```
union REGS {
    struct WORDREGS x;
```

```

        struct BYTEREGS h;
    };

    struct WORDREGS {
        unsigned int ax;
        unsigned int bx;
        unsigned int cx;
        unsigned int dx;
        unsigned int si;
        unsigned int di;
        unsigned int cflag;
    };

    struct BYTEREGS {
        unsigned char al, ah;
        unsigned char bl, bh;
        unsigned char cl, ch;
        unsigned char dl, dh;
    };

```

The 16-bit processor registers AX to ES are represented by the unsigned int variables of the same name in the WORDREGS structure. The 8-bit processor registers AL to DH are represented by the variables in the BYTEREGS structure.

The variant record applies to the 8-bit registers, which are as important as the 16-bit registers for carrying information during the interrupt call. Dividing the 8-bit and 16-bit registers into two variants results in an overlapping of both register sets in memory, with two 8-bit variables overlapping "their" 16-bit variable. So, AL and AH share the same memory space as AX BL, and BH share the same memory space of BX. This also applies to the CL/CH and DL/DH variables.

Notice the order in which 8-bit registers are specified. This order must mirror the format in which the 16-bit register is placed in memory above them. Since, in memory, the low byte of a word precedes the high byte, the L register must be declared before the corresponding H register.

If pregs is a variable of the REGS type, you can easily address the processor registers from the various components of this variable:

```

➤ pregs.x.ax,      ➤ pregs.x.bx,      ➤ pregs.x.cx,
➤ pregs.h.ah,      ➤ pregs.h.dl, etc.

```

If you want to pass the value D3H (0xD3) to the DL register during an interrupt call, do the following:

```
pregs.h.dl = 0xD3;
```

Before calling an interrupt using Intr or MsDos, load the registers, which are used by the function you'll call, with the information you want passed to the function. The interrupt ignores all other registers except those on which it directly relies.

Including the segment register

As the definitions of BYTEREGS and WORDREGS show, these structures ignore the various segment registers and duplicate only the general registers. This occurs because segment registers aren't needed in most function calls. If a function call requires a segment register, use the int86x() and intdosx() functions, which expect a pointer to a variable of the SREGS type, as well as two pointers to variables of the REGS type. The two functions load the various segment registers from this variable before the interrupt call, and save their contents there after the interrupt call.

Here's the definition of SREGS:

```

struct SREGS {
    unsigned int es;
    unsigned int cs;
    unsigned int ss;
    unsigned int ds;
};

```

Reading the flags in the flag register

In many cases, the flag registers can also return information to the calling program. DOS functions extensively use the carry flag, which is set after the function is called and when the function call fails. To simplify checking the flags, the WORDREGS structure contains a field named CFLAG, which is loaded with the contents of the carry flag after a function call. This field shows a value of 1 if the carry flag is set and a value of 0 if it isn't set. Before the function call, the contents of this variable are ignored because the carry flag isn't important for working with interrupt functions until after the interrupt call.

The following program excerpt shows that after the interrupt call it's easy to determine whether the carry flag is set. This program also demonstrates that it's definitely possible to specify a single variable for the inregs and outregs parameters, which will be loaded before the interrupt call with the desired parameters, and accept the contents of the processor register afterwards.

```

#include <dos.h>

void test( void )
{
    union REGS pregs;
    pregs.h.ah = 0x13;          /* Function number */
    pregs.h.dl = 0;             /* Any value */
    intdos( &pregs, &pregs );
    if ( pregs.x.cflag )
        ;                      /* Carry flag set */
    else
        ;                      /* Carry flag unset */
}

```

However, you'll encounter problems if you want to read other flags because some BIOS functions use the zero flag for returning information. In these instances, you can't accomplish anything on Microsoft compilers with the int...() functions. However, the developers at Borland were clever enough to expand the WORDREGS structure by a FLAGS variable, which reflects the contents of the entire flag register after the function call.

```

struct WORDREGS {
    /* Borland only! */
    unsigned int ax, bx, cx, dx, si, di, cflag, flags;
};

```

With Borland compilers, you can determine whether one of the flags is set in the flag register. This is done after calling an int...() function through a binary combination of the flags variable with the value of the particular flag. The table on the right shows the values of the various processor flags.

Buffers and the C language

Many functions expect pointers to buffers when they're called. The functions either take information from the buffers or place information in the buffers (e.g., file contents). These pointers are always FAR; they consist of a segment address and an offset address. This FAR pointer data can be anywhere in memory; it doesn't have to be in the current program's memory segment.

Constant	Bit Pos.	Bit Value
Carry	0	1
Parity	2	4
Auxiliary	4	16
Zero	6	64
Sign	7	128
Overflow	11	2048

Passing pointers to interrupt functions

DOS function 09H (0x09) is an example of a function that takes pointers. This function displays a string on the screen beginning at the current cursor position. Like all DOS functions, it expects the function number in the AH register and the address of the buffer containing the string to be displayed in the DS:DX register pair. DS takes the segment address of the buffer and DX takes the offset address.

Although creating a string is easy in C, you may also want to know how to pass the buffer address. At first this may seem quite simple because both the Borland and the Microsoft compilers define two macros named FP_SEG() and FP_OFF(), which help determine a segment and offset address. However, because these manufacturers define FP_OFF() and FP_SEG() differently, there are some problems:

Borland:

```
#define FP_SEG(fp) ((unsigned)(void _seg *) (void far *) (fp))
#define FP_OFF(fp) ((unsigned)(fp))
```

Microsoft:

```
#define FP_SEG(fp) (*((unsigned _far *)&(fp)+1))
#define FP_OFF(fp) (*((unsigned _far *)&(fp)))
```

Although the Borland definition's macros can contain the variables whose segment or offset addresses you want to determine, the Microsoft macros must be passed a FAR pointer that refers to the appropriate variable.

The following programs also show the differences. Both programs use DOS function 09H (0x09) to display the string from the Message variable on the screen. Unlike other DOS functions, function 09H looks for a \$ character, instead of a null byte, at the end of the buffer. The Borland version is as follows:

```
/****** 9HDEMOBC.C *****/

#include <dos.h>                /* Borland Version */

void main( void )
{
    union REGS pregs;
    struct SREGS sregs;
    char Message[20] = "PC Intern$";

    pregs.h.ah = 0x09;
    sregs.ds = FP_SEG( Message ); /* Get the var. */
    pregs.x.dx = FP_OFF( Message ); /* addresses */
    intdosx( &pregs, &pregs, &sregs );
}
```

As you can see, the FP_SEG() and FP_OFF() functions specify the address at which the message can be found. However, the Microsoft version of the same program requires a FAR pointer that points to the string:

```
/****** 9HDEMOMC.C *****/

#include <dos.h>                /* Microsoft Version */

void main( void )
{
    union REGS pregs;
    struct SREGS sregs;
```

```

char Message[20] = "PC Intern$";
void far *mesptr = Message; /* FAR ptr to string */

pregs.h.ah = 0x09;
sregs.ds = FP_SEG( mesptr ); /* Pass address to */
pregs.x.dx = FP_OFF( mesptr ); /* FAR pointer */
intdosx( &pregs, &pregs, &sregs );
}

```

Receiving pointers from interrupt functions

The following program calls DOS function 1BH (0x1B), which returns a pointer in the DS:BX register pair. This pointer points to a byte containing the media code of the current drive. DOS uses the media code to describe the different types of drives, with codes between F0H (0xF0) and FFH (0xFF). The value F8H (248) characterizes all types of hard drives.

A FAR pointer, from which the media ID can be read, must be generated. Borland implementations of C provide the MK_FP() macro defined in the DOS.H include file. This macro expects two parameters describing the segment and offset addresses to which the desired pointer should refer. This pointer results from this macro and the void far type *.

Although the Microsoft compilers don't define this type of macro, you could easily make your own MK_FP, as shown in the following program listing. MK_FP is defined here, in case it hasn't already been defined by the include files.

After the interrupt call, a FAR pointer is formed by MK_FP() and assigned the mp variable. In the printf() call at the end of the program, this pointer "de-references" the media ID so it can be displayed on the screen.

```

/***** M E D I A I D C . C *****/

#include <dos.h>
#include <stdio.h>

#ifndef MK_FP
/* Macro MK_FP already defined */
#define MK_FP(seg,ofs) ((void far *) ((unsigned long) (seg)<<16|(ofs)))
#endif

void main( void )
{
    union REGS pregs;
    struct SREGS sregs;
    unsigned char far *mp;

    pregs.h.ah = 0x1B;
    intdosx( &pregs, &pregs, &sregs );
    mp = MK_FP( sregs.ds, pregs.x.bx );
    printf( "Media ID = %d\n ", *mp );
}

```

If you examine the definition of the MK_FP() macro, you'll notice that it's quite simple despite the many parentheses and keywords. Within the definition, the segment is cast into a long type, shifted to the left by 16 bits and the offset address is then set in the lower 16 bits of the resulting new long type. The result corresponds exactly to the desired FAR pointer in its composition, so it only has to be accessed by a cast.

Port access in C

Both the Microsoft and Borland compilers offer various functions for accessing ports. However, they have different names and are declared in different include files. Borland has its declarations in DOS.H, while Microsoft has its declarations in CONIO.H. Port Access:C language

The following shows the different routines and declarations of the two compiler manufacturers:

Microsoft: Include file <conio.h>

```
int      inp( unsigned port );
unsigned inpw( unsigned port );
int      outp( unsigned port, int databyte );
unsigned outpw( unsigned port, unsigned dataword );
```

Borland: Include file <dos.h>

```
int      inport (int __portid);
unsigned char inportb(int __portid);
void      output (int __portid, int __value);
void      outportb(int __portid, unsigned char __value);

#define inp(portid)      inportb(portid)
#define outp(portid,v)  outportb(portid,v)
```

As you can see, theoretically the same functions are available from both manufacturers. Each has two functions for reading and writing to ports; one is for 8-bit ports and one is for 16-bit ports.

The Borland compiler demonstrates some cooperation with Microsoft with its inp() and outp() macros, which the Borland functions copied from the names of the two Microsoft functions. Unfortunately, Borland did this only for the two 8-bit functions. However, it's easy to do the same for the two 16-bit functions within a program:

```
#ifdef __TURBOC__
#define inpw(portid)      inport(portid)
#define outpw(portid,v)  outport(portid,v)
#endif
/* Compiling with Turbo C? */
```

This enables you to use the names of the Microsoft functions in your programs even if you're working with a Borland compiler. For example, the following statements read the contents of port 3C4H (0x3C4), which is part of the graphics controller on an EGA/VGA card:

```
XByte = inp( 0x3C4 );
XWord = inpw( 0x3C4 );
```

The following statements allow you to send a byte or word as easily:

```
outp( 0x3C4, XByte );
outpw( 0x3C4 XWord );
```

You'll find examples of these statements in Chapter 4.



The BIOS (Basic Input/Output System)

Most users associate the term operating system with DOS. However, DOS isn't the only operating system on a PC. Before hard drives became standard equipment, the PC searched the BIOS (Basic Input/Output System) for the basic input and output routines needed for communicating between software and hardware. The BIOS is located on a ROM chip that is usually placed on the PC's main circuit board. The BIOS is accessed every time you switch on your PC.

This BIOS contains all the essential routines needed by the PC for communication between hardware and peripheral devices. These routines include instructions for handling screen output, printed output, fonts, date and time.

Why the BIOS is important

Since these routine calls are standardized, the programmer doesn't have to fit programs to one particular PC hardware configuration. This means you can develop a program on one PC or compatible, and run it on another compatible PC without errors, although neither the hardware nor the individual BIOS routines are completely compatible.

The BIOS is an integral part of the PC. It doesn't matter whether a system contains a 20 megabyte hard drive or a 20 Gigabyte hard drive or whether the system is made by IBM or a smaller manufacturer, the BIOS hard drive functions are identical in both instances. This hardware independent concept is mainly responsible for the PC's popularity. It enables computer manufacturers to develop PCs that aren't identical to a true IBM PC but that can still run popular software. Except for additions to accommodate the AT system, few changes have been made to the BIOS since the PC's introduction on the market.

Several companies, including AMI, Phoenix, Award, and Quadtel, manufacture BIOS chips. Although each BIOS has certain differences, they all perform the same essential tasks.

The BIOS Standard

Let's begin with the basics of BIOS: How it works, its ground level functions and how it contributes to starting your PC. IBM defined the types of different BIOS functions and parameters needed in a PC. There are 256 BIOS interrupts, which are divided into functions. This provides a wider selection than that provided by one function per interrupt. These functions provide the communication with the hardware. The table on the right shows the different BIOS interrupts. BIOS views some interrupts as variables, such as video and hard drive functions. (We'll discuss these in more detail later.)

Number	Meaning
10H	Video card access
11H	Configuration test
12H	RAM test
13H	BIOS disk functions
14H	Serial interface functions
15H	Cassette and extended AT functions
16H	Keyboard functions
17H	Parallel interface functions
1AH	Date/time/realtime clock functions

BIOS architecture

The BIOS itself is located in PC ROM, which makes it resident even after the computer has been switched off. It's stored very high in the processor's address space. The ROM chip that contains the BIOS code is always located in the highest area of memory segment F000H.

The exact starting location of BIOS varies depending on the BIOS, the system, and sometimes the memory capacity. For example, the original IBM BIOS started at offset address E000H, while Phoenix BIOS may start at offset address C000H.

The starting point of the BIOS ROM varies with the size of the BIOS ROM. It usually ends at the last memory location of the F segment, at offset address FFFFH. This is the last memory address accessible to Intel processors running in real mode. Some manufacturers add little extras to their BIOS designs so they can beat their competition. For example, VGA cards often bypass ROM-BIOS. These cards include such features as shadow RAM, hard drive parameters, independent setup and password protection. Let's examine these items individually.

Shadow RAM

Shadow RAM is hidden at the same memory addresses as ROM-BIOS. Since double memory allocation isn't permitted in RAM, the ROM-BIOS keeps this shadow RAM hidden from the operating system and applications. Many BIOS systems copy their ROM-BIOS code to shadow RAM, from which BIOS data is accessed. This normally improves execution speed in the PC because the shadow RAM data bus is 16 bits wide and the ROM-BIOS data bus is only 8 bits wide. NEAT chips from Chips & Technologies support shadow RAM.

Hard drive parameters

BIOS often has trouble communicating with the many hard drives on the market. This problem is caused by the different type numbers assigned to each hard drive. Before BIOS can communicate with a hard drive, it must know the number of tracks and sectors available, the number of sectors per track and other hard drive data.

The original solution to this problem was a table of hard drives from which the user could select the drive information using a SETUP program. This information would then be passed to the ROM-BIOS. However, because of the many hard drives on the market, this solution has become obsolete. Instead, the user can manually enter the hard drive parameters using the SETUP program. This information is then passed to the CMOS (Complimentary Metal Oxide Semiconductor) for access from BIOS.

SETUP

SETUP allows the user to configure elements of the ROM-BIOS according to his/her own needs. These elements include date, time and drive types. Some BIOS systems offer the option of configuring part of RAM as expanded memory if the PC supports EMS. Laptop computers include an option for blanking LCD screens and disabling hard drives after a period of keyboard inactivity. This saves battery power on the laptop.

Most SETUP programs support adjustments to the processor's timer frequency. The user can make this adjustment by holding the **[Alt]** key and pressing the **[+]** and **[-]** keys on the numeric keypad.

Password entry

ROM-BIOS is the best place to include BIOS password protection. Password access can then be requested before the system starts, and before DOS is loaded. Many BIOS manufacturers permit password entry through the SETUP program. The password is then stored in CMOS (battery operated RAM) or on the hard drive.

POST

Program execution in a computer based on the Intel 8088 (or one of its successors) starts after the computer is switched on at memory location F000:FFF0. This memory location is part of the ROM-BIOS and contains a jump instruction to a BIOS routine, which takes system testing and hardware component initialization. This routine is called the POST (Power-On Self-Test).

POST tests

The POST consists of many tests for checking onboard PC hardware (the processor, memory, the interrupt controller, DMA, etc.), as well as the ability to initialize expansion cards (e.g., video cards). If an error occurs during these tests, the POST displays an error message or error number on the screen and instructs the computer to beep.

3. The BIOS (Basic Input/Output System)

65

The following list shows the different tests performed by the POST and the sequence in which these tests are executed. This sequence isn't absolute and can change depending on the manufacturer.

- Function check of CPU (coprocessor, real mode, protected mode, etc.)
- BIOS ROM checksum
- CMOS RAM (battery operated RAM) checksum
- Test/initialize DMA controller
- Test/initialize keyboard controller
- Check first 64K of RAM
- Test/initialize interrupt controller
- Test/initialize cache controller (AT only)

First, the POST tests individual functions of the processor, its registers, and some instructions. If an error occurs during this test, the system stops without displaying an error message (since the processor is defective, screen display would be impossible). If the processor passes the test, a checksum is computed for each of the BIOS ROM's contents and compared with the various ROMs to determine whether a defect exists there. Each chip on the main circuit board (such as the 8259 interrupt controller, the 8237 DMA controller, and the RAM chips) undergoes tests and initialization.

- Video controller
- Serial and parallel interfaces
- RAM above 64K
- Disk and hard drive controllers

Peripheral testing

After determining that the main circuit board is fully functional, the POST tests the peripherals (keyboard, disk drives, etc.). Besides these hardware related tasks, the BIOS variables and the interrupt vector table must be initialized.

Searching for ROM extensions

Once these tests are completed, the search for ROM extensions begins. These ROM extensions originate either from the main circuit board or an expansion card, and augment or replace onboard BIOS functions. For example, EGA, VGA, and Super VGA cards have their own BIOS functions to replace the old BIOS interrupt 10H, which was designed specifically for handling MDA and CGA cards. Also, SCSI controllers, which are used for controlling hard drives, don't use BIOS disk interrupt 13H.

The POST tests for ROM extensions by checking offset 00H and 01H in the memory range allocated for BIOS functions. A BIOS extension exists if the contents of these two bytes are 55H and AAH respectively. Offset 02H indicates the size of the ROM module in blocks of 512 bytes. The module's initialization routine begins at offset 03H.

Initialization of ROM Module		
Offset	Contents	Type
00H	ID byte #1 (55H)	1 byte
01H	ID byte #2 (AAH)	1 byte
02H	Module length in 512-byte blocks	1 byte
	Initialization routine

ROM modules

These ROM modules have the option of replacing existing BIOS routines with their own routines, and integrating these new routines with the system. The module routines must be placed in memory ranges specifically allocated for such routines.

C000H:0000H - C000H:7FFFH: EGA and VGA BIOS extensions

This range is usually reserved for the BIOS extensions provided by EGA, VGA, and Super VGA cards. BIOS divides this range into 2K increments because most extensions accept this division.

C000H:8000H - D000H:FFFFH: Hard drive extensions

This range is usually reserved for the BIOS extensions provided by many hard disk controllers. BIOS divides this range into 2K increments. The D segment of this range (D000H:0000H - D000H:FFFFH) is often used for the page frame by EMS cards. If an EMS card is being used, this range is unavailable for ROM extensions.

E000H:0000H - E000H:FFFFH: Miscellaneous

This range is reserved for BIOS systems that require more memory than is provided by memory segment F. Few BIOS extensions recognize this range.

After POST

Once ROM initialization ends, the boot process directly applying to BIOS also ends. Interrupt 19H, known as the bootstrap loader, tries to load some form of the basic operating system on startup or on system reset (when you press the **Alt** **Ctrl** **Del** key combination), from a predetermined place on the diskette.

This bootstrap process may fail for various reasons:

- There is no disk in the disk drive.
- There is a non system disk in the drive (the DOS files are not available on the diskette). If this occurs, the bootstrap routine attempts to find the routine on the other disk drives connected to the PC, or on a predetermined location on an existing hard disk.

If the system still cannot find the bootable system disk, there are two other reasons that may be causing the problem:

- Some older systems switch to ROM BASIC, a BASIC interpreter stored in PC ROM directly beneath the BIOS, starting at memory location F000H:6000H. Newer PCs display a message on the screen requesting that the user insert a system diskette and press a key.
- BIOS doesn't care what operating system it loads, so it may attempt to load a non-DOS operating system if one exists on the disk. This makes it possible to load other operating systems, such as XENIX.

Determining BIOS Version

Next to the BIOS code and some static variables (e.g., the hard drive parameter table), you'll find information describing the BIOS brand and the type of PC. You can access this information.

The previous section described memory location F000H:FFF0H with the system startup and POST. A 5-byte long jump instruction to the POST routine is usually found at this location. After this instruction, an additional 11 bytes are available (to F000:FFFF) in the ROM chip normally used to store the BIOS version or release date. You can examine the contents of these memory locations to determine which BIOS version your PC uses. Call the DEBUG program from the DOS prompt:

```
debug
```

Enter the following line to display the bytes at the end of the ROM-BIOS (the character following the memory location is a lowercase "l", not the number "1"):

```
d f000:fff0 l 10
```

The next line displays the contents of this memory location as a hexadecimal number; the characters to the right of the hex display are the corresponding ASCII codes. Day, month and year appear as two digits separated by "/" characters.

*BIOS date
display in
DEBUG*

```
C>debug
-d f000:fff0 l 10
F000:FFF0 EA 5B E0 00 F0 30 32 2F-30 36 2F 38 36 00 FC 00 [...02/06/86...]
-q
C>_
```

Determining the PC Type

Certain BIOS functions are used more for model identification than for BIOS version identification. They indicate the type of PC being used. They also indicate when the BIOS has additional functions (e.g., AT BIOS is better equipped than the PC and XT BIOS). These extra functions essentially handle string output on the screen, realtime clock access (standard on the AT), and additional RAM beyond the 1 megabyte memory limit (also standard on the AT).

Model identification byte codes	
Code	Meaning
FCH	AT
FEH FBH	XT
FFH	PC

A program that calls these functions must first ensure that the computer being used is actually an AT, and that the functions addressed are available. The programmer can use the model identification byte located in the last memory location of the ROM-BIOS at address F000:FFFE. This byte can contain the codes listed in the table on the left.

These values aren't entirely accurate. Many PC/XT compatibles indicate completely different values in the model identification byte. Use the following guideline: A model identification byte of FCH identifies an AT; any other number indicates a PC/XT.

Only IBM computers have guaranteed reliable model identification numbers at memory location F000:FFFE. This may not apply to compatibles because the BIOS varies slightly with each manufacturer. Chapter 16 discusses processor types in more detail.

BIOS Variables

The preceding sections described different BIOS interrupts and their functions. These functions require a segment of memory for storing variables and data. Therefore, the BIOS variable memory reserves over 256 bytes of memory, starting at address 0040H:0000H, for storing internal variables. This range is called the *BIOS variable range* or *BIOS variable segment*.

This memory range's allocation is standardized because many DOS programs directly access the BIOS variables and BIOS manufacturers don't provide alternate ways of accessing these variables. This standardization refers to the BIOS variables developed for the PC and PC/XT models, which stands in this range up to offset address 0071H. Memory beyond this point is used by EGA and VGA cards, as well as AT and PS/2 systems. The contents change after 0071H depending on the BIOS, PC type, and video card available.

The following list describes the individual variables, their purposes, and addresses. The address indicated is the offset address of segment address 0040H. For example, a variable with the offset address 10H has the address 0040H:0010H or 10H.

00H	Serial interface port addresses	4 words	INT 14H
-----	---------------------------------	---------	---------

During the POST (Power On Self Test), a BIOS routine determines the configuration of its PC. Among other things, this routine determines the number of installed serial (RS-232) interfaces. These interface numbers are stored as four words in memory. Each word represents one of the four cards that can be installed for asynchronous data transmission. First the low byte is stored, followed by the high byte. Since few PCs have four serial cards at their disposal, the words that represent a missing card contain the value 0.

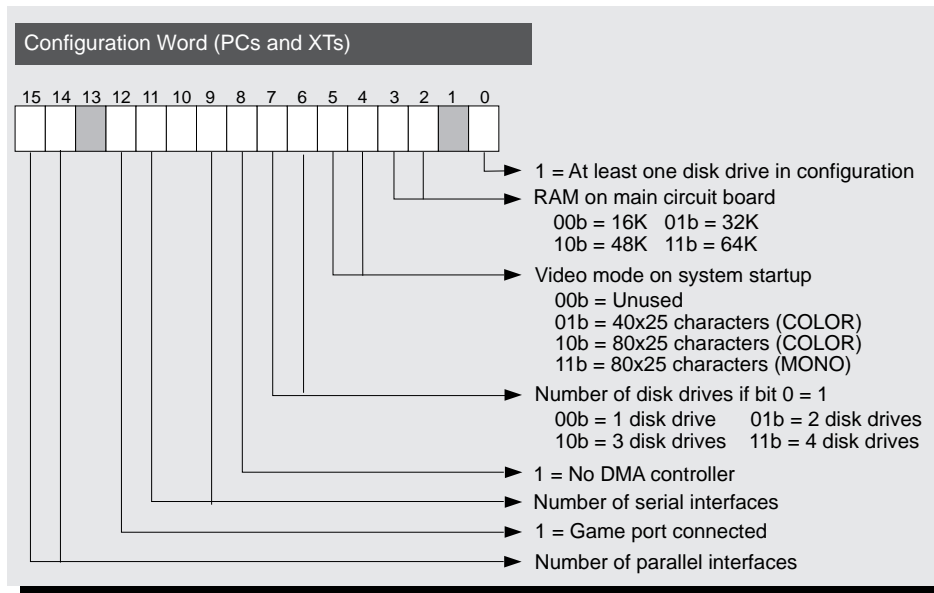
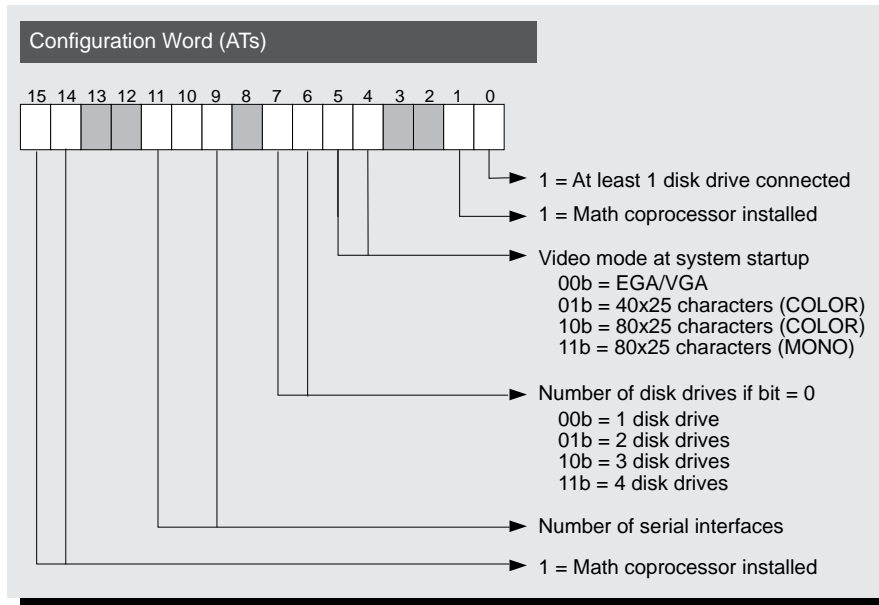
08H	Parallel interface port addresses	4 words	INT 17H
-----	-----------------------------------	---------	---------

During the POST (Power On Self Test), a BIOS routine determines the configuration of its PC. It determines the number of installed parallel interfaces. These card numbers are stored as four words in memory. Each word represents one of the four

cards that can be installed for parallel data transmission. First the low byte is stored, followed by the high byte. Since few PCs have four parallel cards at their disposal, the words that represent a missing card contain the value 0.

10H	Configuration	1 word	INT 11H
-----	---------------	--------	---------

This word represents the hardware configuration of the PC as called through BIOS interrupt 11H. Similar to the previous two words, this configuration is determined during the booting process. The purposes of individual bits of this word are standardized for the PC and the XT, but can differ in other computers.



3. The BIOS (Basic Input/Output System)

69

12H	POST status #1	1 byte	POST
-----	----------------	--------	------

This byte provides storage for information gathered during the POST, and executed during the booting process and after a warm start. BIOS routines also use this byte for recognizing active keys. It has no practical use for the programmer.

13H	RAM size	1 word	INT 12H
-----	----------	--------	---------

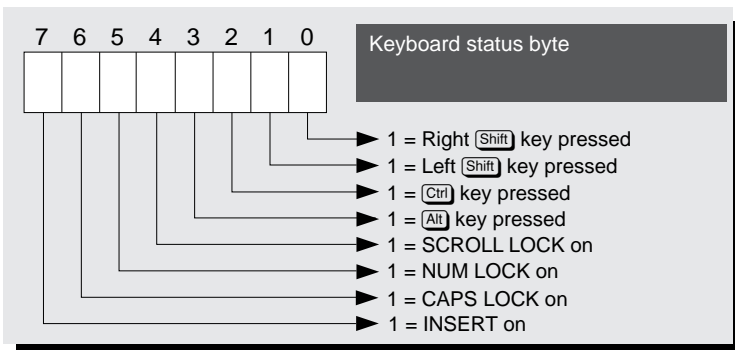
This word indicates the RAM capacity of the system in kilobytes (not counting expanded memory). This information is also gathered during the booting process, and can be read using BIOS interrupt 12H.

15H	POST status #2	1 word	POST
-----	----------------	--------	------

These two bytes test the hardware during the booting process. How this test is performed varies with the BIOS.

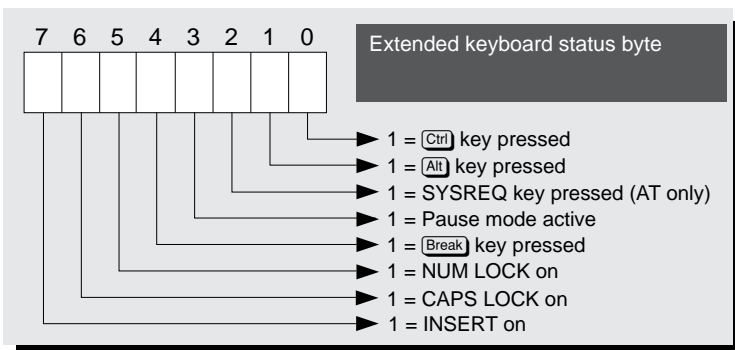
17H	Keyboard status byte	1 byte	INT 16H
-----	----------------------	--------	---------

This is called the keyboard status byte because it contains the status of the keyboard and different keys. Function 02H of BIOS keyboard interrupt 16H reads this byte. Accessing this byte allows the user to toggle the **Ins** or **Caps Lock** key on or off. The upper four bits of this byte may be changed by the user; the lower four bits must remain undisturbed.



18H	Extended keyboard status byte	1 byte	INT 16H
-----	-------------------------------	--------	---------

This byte is similar to byte 17H above, except that this byte indicates the active status of the **Sys Req** and **Break** keys. Bit 3 indicates the status of pause mode.



19H	ASCII code entry	1 byte	INT 16H
-----	------------------	--------	---------

This byte isn't used in older systems. Newer systems use this byte for storing ASCII codes produced from the numeric keypad and the **Alt** key.

1AH	Next character in keyboard buffer	1 word	INT 16H
-----	-----------------------------------	--------	---------

This word contains the offset address of the next character to be read in the keyboard buffer (see also 1EH).

1CH	Last character in keyboard buffer	1 word	INT 16H
-----	-----------------------------------	--------	---------

This word contains the offset address of the last character in the keyboard buffer (see also 1EH).

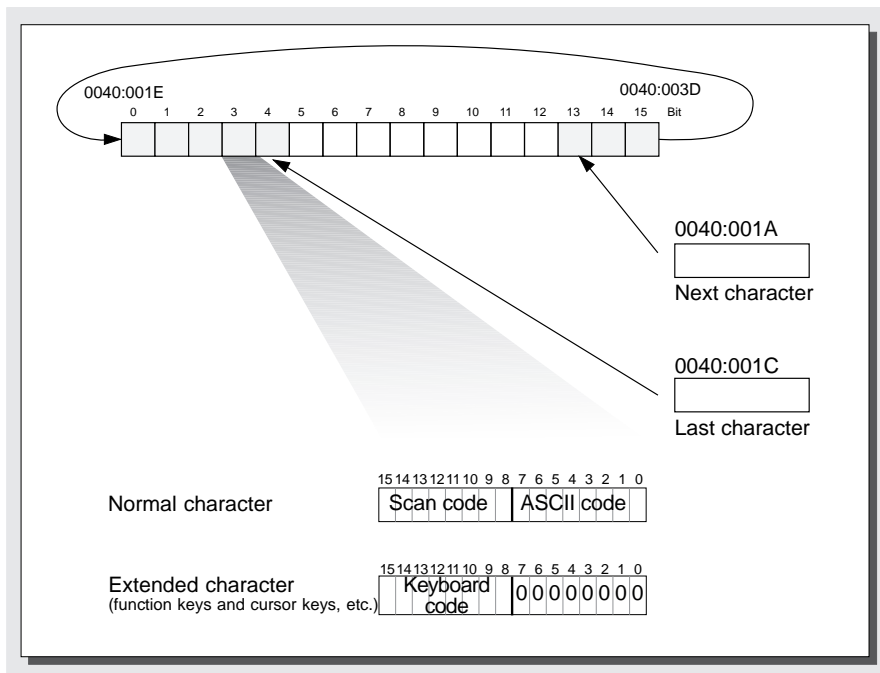
1EH	Keyboard buffer	16 words	INT 16H
-----	-----------------	----------	---------

These 16 words contain the actual keyboard buffer. Since every character stored in the keyboard buffer requires 2 bytes, its 32-byte capacity provides space for a maximum of 16 characters. For a normal ASCII character, the buffer stores the ASCII code and then the character's scan code (the number of the key that generated the ASCII character). If the character in the keyboard buffer uses an extended code (e.g., a cursor key), then the first byte contains the value 0 and the second byte contains the extended key code.

The computer constantly reads characters from the keyboard buffer. If the buffer isn't full, characters can be added. The address of the next character to be read from the keyboard buffer is stored in the word at offset address 001AH. When a character is read, the character moves 2 bytes toward the end of the buffer in memory. When a character was read from the last memory location of the buffer, this pointer resets to the beginning of the buffer. This also applies to the pointer in offset address 001CH, which indicates the end of the keyboard buffer. If you add a new character, it is stored in the keyboard buffer at the location indicated by this pointer. Then the pointer is incremented by 2, moving toward the end of the buffer. If a new character is stored at the last memory location of the buffer, this pointer resets to the beginning of the buffer.

The relationship between the two pointers is an indication of the buffer's status. Two conditions are especially important. In one condition, both pointers contain the same address (no characters are currently available in the keyboard buffer). In the other condition, a character should be appended to the end of the keyboard buffer, but adding 2 to the end pointer would point it to the start pointer. This means the keyboard buffer is full, (no other characters can be accepted).

*Keyboard buffer
with start, end
pointers and
ring buffer*

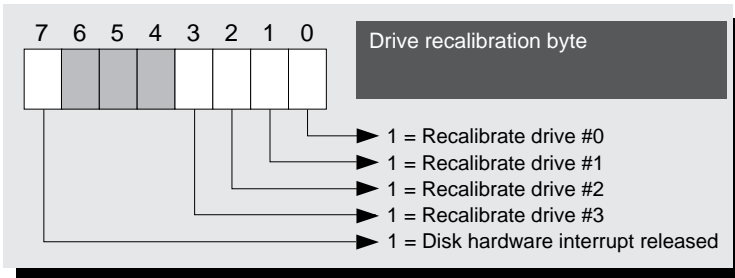


3. The BIOS (Basic Input/Output System)

71

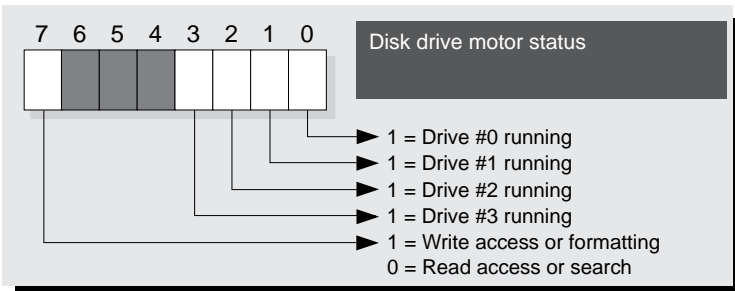
3EH	Disk drive recalibration	1 byte	INT 13H
-----	--------------------------	--------	---------

The lowest four bits correspond to the number of installed PC disk drives specified by BIOS (you can use a maximum of four drives). These bytes also indicate whether the connected drives must be calibrated. Usually this is necessary after an error occurs during read, write, or search access. Bit 7 is set to 1 when a disk drive releases the disk hardware interrupt.



3FH	Disk drive motor status	1 byte	INT 13H
-----	-------------------------	--------	---------

The four lower bits of this byte indicate whether the current disk drive motor is running. A 1 in the corresponding bit indicates this. Bit 7 is always set during write access or formatting and unset during read access or a search.



40H	Disk drive motor timer	1 byte	INT 13H
-----	------------------------	--------	---------

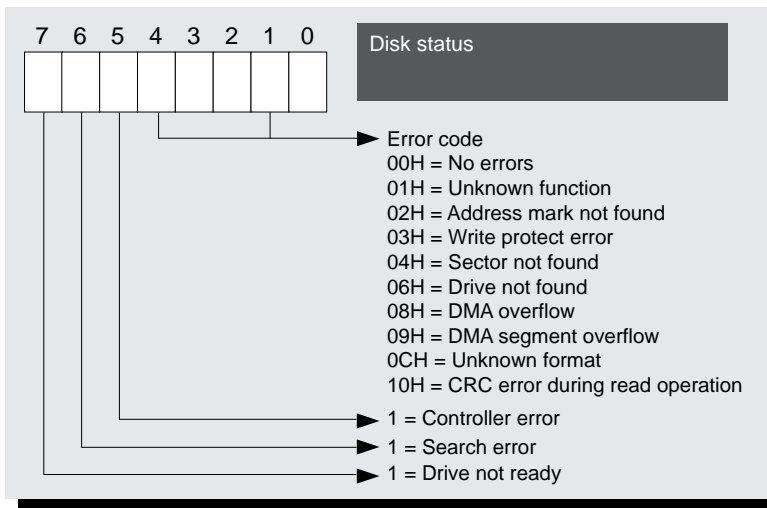
This byte contains a numerical value that indicates the number of calls made to the timer (interrupt 08H) until a disk drive motor switches off. Since BIOS can only access one disk drive at a time, this value refers to the last drive that was accessed. Following access to this drive, BIOS places the value 37 (25H) into this register, indicating a duration of about two seconds. During each timer interrupt (which occurs about 18.2 times per second), the value in this byte is decremented by 1. When it finally reaches 0, the disk motor is switched off. This occurs after about two seconds.

41H	Disk error status	1 byte	INT 13H
-----	-------------------	--------	---------

This byte contains the status of the last disk access. When the byte contains the value 0, the last disk operation was performed in an orderly manner. Another value signals that an error code was transmitted by the disk controller.

42H	Disk controller status	7 bytes	INT 13H
-----	------------------------	---------	---------

These seven bytes indicate the status of the disk controller. They also indicate hard disk controller status on hard disk systems.



49H	Current video mode	1 byte	INT 10H
-----	--------------------	--------	---------

This byte contains the current video mode as reported by the BIOS. This is the same value indicated when the user activates a video mode through function 0H of BIOS video interrupt 10H.

4AH	Number of screen columns	1 word	INT 10H
-----	--------------------------	--------	---------

This word contains the number of text columns per display line in the current display mode.

4CH	Screen page size	1 word	INT 10H
-----	------------------	--------	---------

This word contains the number of bytes required for the display of a screen page in the current display mode, as reported by the BIOS. In the 80x25 character text mode, this is 4,000 bytes.

4EH	Offset address of current screen page	1 word	INT 10H
-----	---------------------------------------	--------	---------

This word contains the address of the current screen page now on the monitor, relative to the beginning of video RAM.

50H	Cursor position in eight screen pages	8 words	INT 10H
-----	---------------------------------------	---------	---------

These 8 words contain the current cursor position for each screen page. BIOS can control a maximum of 8 screen pages and reserves two bytes for each screen page. The low byte indicates the screen column; the high byte indicates the screen line.

60H	Starting line of screen cursor	1 byte	INT 10H
-----	--------------------------------	--------	---------

This byte contains the starting line of the blinking cursor, which can have values ranging from 0 to 7 (color card) or from 0 to 14 (monochrome card). Changing the contents of this byte doesn't change the cursor's appearance, because first it must be transmitted by BIOS to the video controller.

61H	Ending line of screen cursor	1 byte	INT 10H
-----	------------------------------	--------	---------

This byte contains the ending line of the blinking cursor, which can have values ranging from 0 to 7 (color card) or from 0 to 14 (monochrome card). Changing the contents of this byte doesn't change the cursor's appearance, since it must first be transmitted by BIOS to the video controller.

62H	Current screen page number	1 byte	INT 10H
-----	----------------------------	--------	---------

This byte contains the number of the currently displayed screen page.

3. The BIOS (Basic Input/Output System)

73

63H	Port address of video controller	1 word	INT 10H
-----	----------------------------------	--------	---------

This word contains the address of the video card port. If a PC contains several video cards, the value stored will be the address of the currently active video card's port. This address is 3B4H in monochrome video cards, and 3D4H on CGA, EGA, and VGA video cards.

65H	Mode selector register contents	1 byte	INT 10H
-----	---------------------------------	--------	---------

The contents of a video controller card's mode selector determines the current video mode. The current value is stored in this memory location.

66H	Palette register contents	1 byte	INT 10H
-----	---------------------------	--------	---------

A color card in medium-resolution CGA compatible graphic mode can display 320x200 pixels in four different colors. This byte indicates the currently active color palette.

67H	Miscellaneous	5 bytes	POST
-----	---------------	---------	------

The early PC BIOS versions could use a cassette recorder for data storage. Those early versions of BIOS used these five bytes for cassette access when storing data. XT and AT models, which don't have this interface, use these memory locations for other purposes.

6CH	Timer	1 dword	INT 1AH
-----	-------	---------	---------

These four bytes act as a 32-bit counter for both BIOS and DOS. The counter is incremented by 1 on each of the 18.2 timer interrupts per second. This permits time measurement and time display. The value of this counter can be read and set with BIOS interrupt 1AH. If 24 hours have elapsed, it resets to 0 and counts up from there.

70H	24-hour flag	1 byte	INT 1AH
-----	--------------	--------	---------

This byte contains a 0 when the timer routine is between 0 and 24 hours. Byte 70H changes to 1 when the time counter routine exceeds its 24 hour limit. If the BIOS timer interrupt 1AH is used to set the time, this byte resets to 0.

71H	CTRL-Break flag	1 byte	INT 16H
-----	-----------------	--------	---------

This byte indicates whether a keyboard interrupt occurs after the user presses **Ctrl****C** or **Ctrl****Break**. If bit 7 of this byte contains the value 1, a keyboard interrupt has occurred.

XT BIOS variables

The hardware configurations of the XT permit the introduction of additional variables. The following is a list of BIOS variables found in the XT and AT.

72H	POST test	1 word	POST
-----	-----------	--------	------

During the POST, a reset command is sent to the keyboard controller, whether a cold or warm start has occurred. For the duration of this reset, this location assumes the value 1234H. No memory test occurs when a warm start is executed.

74H	Last hard drive operation (AT)	1 byte	INT 13H
-----	--------------------------------	--------	---------

This byte indicates the status of the last hard drive operation.

01H	Function not available, or invalid drive specification	0EH	Address mark not found
02H	Address marker not found	0FH	DMA overflow
04H	Sector not found	10H	Read error
05H	Controller reset error	11H	Corrected ECC read error
07H	Controller initialization error	20H	Controller defect
09H	DMA transfer error: Segment overflow	40H	Seek failed
0AH	Bad sector	80H	Drive time out
0BH	Bad track	AAH	Drive not ready
0DH	Invalid number of sectors in track	CCH	Write error

75H	Number of hard drives (AT)	1 byte	INT 13H
-----	----------------------------	--------	---------

This byte indicates the number of hard drives connected to the system.

76H	Hard drive control byte (AT)	1 byte	INT 13H
-----	------------------------------	--------	---------

This byte controls the hard drive from BIOS interrupt 13H. Its exact purpose is unknown.

77H	Hard drive port (AT)	1 byte	INT 17H
-----	----------------------	--------	---------

This byte contains the base address of the hard drive controller.

78H	Parallel interface time out counter	4 bytes	INT 14H
-----	-------------------------------------	---------	---------

These 4 bytes correspond to the time out counters for the four parallel interfaces. Each byte indicates the number of times a parallel time out error occurs.

7CH	Serial interface time out counter	4 bytes	INT 16H
-----	-----------------------------------	---------	---------

These 4 bytes correspond to the time out counters for the four serial interfaces. Each byte indicates the number of times a serial time out error occurs.

80H	Keyboard buffer starting address (AT)	1 word	INT 16H
-----	---------------------------------------	--------	---------

This word contains the beginning of the keyboard buffer as the offset address to segment address 0040H. Since the keyboard buffer normally starts at address 0040H:001EH, this memory location usually contains the value 1EH.

82H	Keyboard buffer ending address (AT)	1 word	INT 10H
-----	-------------------------------------	--------	---------

This word contains the end of the keyboard buffer as the offset address to the segment address 0040H.

84H	Number of screen lines (EGA/VGA)	1 byte	INT 10H
-----	----------------------------------	--------	---------

This byte contains the number of screen lines being used by the EGA or VGA card.

85H	Character height (EGA/VGA)	1 word	INT 10H
-----	----------------------------	--------	---------

This byte indicates EGA/VGA character height in pixels, as well as the number of visible text lines.

87H	EGA/VGA status range (EGA/VGA)	4 bytes	INT 10H
-----	--------------------------------	---------	---------

These 4 bytes indicate the status of the EGA or VGA card.

3. The BIOS (Basic Input/Output System)

75

8BH	Disk drive/hard drive parameters (PS/2) 11 bytes	INT 13H	INT 16H
-----	--	---------	---------

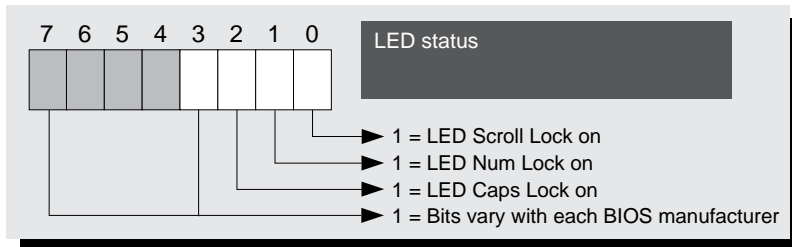
These bytes describe PS/2 disk drive and hard drive information.

96H	MF II status (AT)	1 byte	INT 16H
-----	-------------------	--------	---------

This byte indicates the status of an MF II model keyboard. Bit 4 of this byte indicates whether the system includes an MF II American (101-key) or European (102-key) keyboard. Applications that use the additional keys found on the MF II keyboard (e.g., **F11** and **F12**) check 96H and adjust to the keyboard.

97H	LED status (AT)	1 byte	INT 16H
-----	-----------------	--------	---------

This byte indicates the status of the keyboard LEDs. MF II keyboards include three LEDs, which correspond to the three toggled keyboard modes (Num Lock, Caps Lock, and Scroll Lock). Function 02H returns keyboard status without reading characters from the keyboard.



98H	Wait flag pointer (AT)	1 dword	INT 15H
-----	------------------------	---------	---------

You can define a BYTE variable whose bit 7 will be set to 1 after a specific amount of time has elapsed (see the description of interrupt 15H, function 83H in the Appendices for more information). The address of the BYTE variable is stored at this location in the BIOS variable segment.



You'll find the Appendices on the companion CD-ROM

9CH	Timer (AT)	1 dword	INT 15H
-----	------------	---------	---------

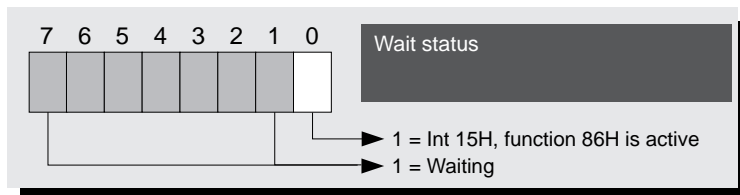
This DWORD represents the variable in which the timer duration can be placed before passing the duration to the caller (see the descriptions of interrupt 15H, functions 83H and 86H in the Appendices for more information).



You'll find the Appendices on the companion CD-ROM

A0H	Wait status (AT)	1 byte	INT 15H
-----	------------------	--------	---------

This variable states whether the system is waiting and whether interrupt 15H, function 86H is active.



A1H	Reserved	95 bytes	-----
-----	----------	----------	-------

This range is reserved for BIOS extensions and programs.

100H	Hardcopy recursion flag	1 byte	INT 05H
------	-------------------------	--------	---------

All PC types have a variable in common, at memory location 0050H:0000H. This location is used by the hardcopy routine (interrupt 05H) as a recursion flag. The recursion flag prevents the user from printing more than one hardcopy at a time. When the hardcopy routine is executing, this flag is set to 1; otherwise it is set to 0. Output errors set this flag to 255.

4

A Closer Look At Video Cards

Since there are several graphic standards (MDA, CGA, EGA, VGA and Hercules), a single standard hasn't been established for video cards. Even the new Super VGA and TIGA card types don't have a single standard. We'll describe all these video graphic cards in this chapter.

History And Highlights

Let's begin with an summary of the history of the different video standards used in PCs. Significant advancements have been made in two areas of computer hardware technology. Processor speeds have increased and video cards have been improved. The video card improvements have resulted in higher resolutions and a larger spectrum of colors.

A few years ago, advancements in video cards dramatically improved the capabilities and performance of video displays. The original idea was to take the burden of drawing lines and figures away from the 80x86 processor. As you probably already know, graphical user interfaces have become the preferred way to interact with the computer. Therefore, video technology has become even more important because software places more demands on the processor. If the application seems to operate quickly, the video card is probably sharing some of the work with the 80x86 processor. Most modern graphic cards with the popular S3 chip or a different graphic processor perform especially well in these situations. (We'll discuss these cards later.)

In the following sections we'll discuss the history of hardware development and describe the highlights of various types of video cards.

Monochrome Display Adapter (MDA)

Besides the CGA card, the IBM Monochrome Display Adapter or MDA, is the oldest graphics adapter available for the PC. The MDA was the standard when IBM released the first PCs in 1981. The MDA card supports only one operating mode. This is a text mode consisting of 80 screen columns and 25 screen rows. Unlike other graphics cards, the MDA contains very little video RAM. So, it can store only one screen page in RAM.

Although this card cannot display graphics, many users preferred the MDA over the CGA card because it was the only alternative at the time. Compared to CGA, the MDA actually has a higher screen resolution, which reduces eyestrain. Few PCs use MDA cards today and IBM stopped manufacturing them years ago. The Hercules Graphics Card (HGC) replaced the MDA in 1982. The Hercules card has all the attributes of the MDA but can also display graphics.

Color/Graphics Adapter (CGA)

The CGA (Color/Graphics Adapter) standard was also introduced in 1981. This card, which can display graphics, offered users an alternative to the MDA card. Users who could afford a CGA card could actually save money. Instead of using a monitor, these users could connect a standard television set to a special connector on the CGA card. Also, a CGA card can produce RGB output, in which electrical lines send different signals for the colors red, green and blue. However, the CGA graphics quality wasn't as good as the MDA's because of the larger three color pixels that were generated.

Similar to the MDA card, the CGA card also has a text mode consisting of 80 columns and 25 rows. The individual characters are based on a smaller pixel matrix. However, a CGA card can also display graphics with a resolution of 320x200 pixels, in four colors. Color suppressed mode produces graphics with a resolution of 600x200 pixels, in only two colors.

Although CGA and MDA differ, they are based upon the same video controller (the Motorola MC6845).

Hercules Graphics Card (HGC)

In 1982, a year after the PC was introduced, Hercules released a graphics card that immediately made them famous. The legendary Hercules Graphics Card (HGC), which was based on the Motorola MC6845, was completely MDA compatible. This card displayed two 720x348 graphic screen pages. The Hercules card combined the readability of the MDA card and the graphic output of the CGA card. However, it also had the resolution to display high quality graphics and text.

Many experts still consider the Hercules card as the standard among monochrome graphics cards. Whenever monochrome cards must be used instead of the more popular color cards, the Hercules card is used. Although CGA and MDA cards are seldom released today, a few manufacturers still produce Hercules Graphics Cards.

Unfortunately both the original HGC and its clones have a flaw. Since IBM won't support third party video cards, Hercules cards have incomplete BIOS support. However, the system tolerates Hercules cards because they are compatible with the old MDA card and because of ROM BIOS support in text mode. When discussing the graphics mode on this card, you must remember that it doesn't support BIOS. This applies to graphics mode initialization and screen pixel access. This isn't actually a problem because the BIOS would only slow down screen display. As you'll see in this chapter, it's easy to access Hercules pixel information.

Since the Hercules Graphics Card represents a fixed standard in the ever-changing PC market, the card has undergone the miniaturization applied to many PC components. While the first Hercules cards required a full card's length and 40 ICs, newer Hercules cards required half a card and use as few as 10 ICs. Some of these cards also included a parallel printer interface. Although the Hercules Corporation has manufactured new video cards (the Hercules Graphics Card Plus and the Hercules InColor Card), they haven't achieved the success of the original Hercules Graphics Card.

Enhanced Graphics Adapter (EGA)

After the release of the Hercules Graphics Card, IBM tried to design a card to replace the CGA card and surpass the capabilities of the Hercules card. The result was the EGA (Enhanced Graphics Adapter), which was released in 1985. Due to the many technological advances that occurred between 1981 and 1985, the EGA started a minor revolution in PC computing. Since the EGA was more powerful than the CGA and MDA cards combined, it set a new standard for screen resolution and price. This card placed high resolution graphics in a price range most users could afford.

The EGA card has its own video modes, as well as fully compatible MDA and CGA modes. This is useful for programs that support multiple video modes. Because of its ability to display monochrome graphics on a monochrome monitor, the EGA is similar to the Hercules card. The EGA card was the first graphics card for the PC that could handle both monochrome and color screens.

The EGA is most effective when it's combined with an EGA monitor. This monitor is similar to a CGA monitor except the graphics mode resolution is much higher (640x350 pixels) and more color options are available (16 colors at a time, from a total palette of 64 colors). Also, the EGA card contains increased video RAM (some EGA cards can hold up to 256K of video RAM) for displaying different graphic screen pages.

Instead of the MC6845 video controller, the EGA card uses highly integrated VLSI chips for handling video display. All screen information is stored in video RAM, which makes this standard dramatically different from earlier methods. Because of its smaller pixel size, the EGA's screen resolution is sharper than the CGA's resolution. Also, the EGA offers more options for generating custom fonts than the earlier cards. The EGA card also gives users the power needed to create computer animation and other applications, such as arcade-style games.

Unlike MDA and CGA cards, the EGA isn't supported by the IBM ROM BIOS. So, the EGA has its own ROM BIOS. The EGA ROM BIOS replaces the original BIOS and allows access to all the features of the EGA card.

As the EGA became more popular, manufacturers began developing compatible cards with additional video modes, which weren't supported by many programs. Even though IBM sued manufacturers for marketing EGA compatible cards, it couldn't stop the flow of compatible cards from the Far East.

Many EGA cards are still being used although VGA cards have replaced EGA cards as the standard for video display. However, many VGA cards include EGA modes.

Video Graphics Array (VGA)

The VGA (Video Graphics Array) card was released in 1987, which was the same year IBM introduced its PS/2 systems. This card combines new technology and the features found in the EGA card. So, it maintains compatibility with all predecessors and offers more colors, higher resolution and better text display.

Although today most VGA cards are inexpensive, the monitor needed for VGA graphics is expensive. Although users may not want to view VGA display in monochrome, many computer systems are equipped with only VGA monochrome monitors. The VGA standard was originally designed for IBM's PS/2 machines and the Micro Channel bus. However, since many manufacturers sell VGA cards for the ISA bus, almost any system can use a VGA card.

The VGA's advantages over EGA is its higher integration density and an entire control logic that's packed into a single chip. Unlike the EGA card, the VGA card sends analog color signals to its monitor instead of digital signals. This means that VGA cards can generate more than 260,000 different colors when modes 2, 4, 16 or 256 are active.

The highest resolution VGA mode provides 640x480 pixels, with either 2, 4 or 16 colors, depending on the mode selected. The extended 320x200 pixel mode is more versatile, offering up to 256 colors on the screen at a time. Higher resolution or more colors means that some video RAM will be needed to handle screen information. So, VGA cards frequently contain a minimum of 256K of video RAM; this can easily be increased to 512K.

Like an EGA card, a VGA card has its own BIOS, which replaces the standard BIOS video output functions. The VGA hardware is often downwardly compatible with EGA BIOS. So, all the programs intended for the EGA BIOS will also operate without problems under the VGA card. Third party VGA cards encounter the same problems faced by the EGA cards (added video modes and different color capabilities). Although it may be tempting for the system programmer to use one of these additional modes or color sets, we'll concentrate on standard VGA modes in this book. An extended VGA standard will make it possible to standardize the extended video modes with access to any program.

Super VGA

Super VGA cards have the same hardware as normal VGA cards, but they display pixels faster, in more colors and with higher resolutions than their predecessors. These cards support all VGA modes. While a normal VGA card can display 256 colors in 320x200 mode, Super VGA cards can display the same amount of colors in three other modes (640x200, 640x350 and 640x480 pixels). Other graphics modes can display 800x600 and 1024x768 pixels on a compatible VGA or multiscan monitor, if sufficient video RAM is available.

Again, different manufacturers have added their own extended modes and hardware registers to Super VGA cards. The largest VGA chip manufacturers (Tseng, Paradise and Video Seven) formed a consortium, called Video Electronic Standard Association (VESA). Its goal was to present a standard for Super VGA modes and video BIOS that was based on the chips developed by these three manufacturers. TSR programs can be used to add the new BIOS functions to older Super VGA cards.

Unfortunately, this consortium wasn't formed until 1990, so a lot of time passed before the VESA standard became effective. Until then, every program had to directly access the hardware of different Super VGA cards to use the extended VGA modes.

Memory Controller Gate Array (MCGA)

While VGA cards were designed for the IBM's upscale PS/2 models, the MCGA; (Memory Controller Gate Array) cards were designed for the lower end PS/2 machines. This card was intended to replace almost every previous standard. The MCGA's text mode, which is similar to the CGA card's, provides a 80x24 character display. The foreground and background colors can be selected from a 16 color palette. Unlike the CGA card, the MCGA's palette can be selected from a group of 262,000 colors (similar to VGA). The MCGA's vertical resolution in text mode is 400 pixels rather than 200 pixels, which provides a higher quality display.

For a hybrid, the MCGA card handles various graphics modes. In addition to two VGA compatible modes, the MCGA supports both CGA modes (320x200 and 640x200 pixels). Because the card uses a vertical resolution of 400 pixels, the vertical pixels in the CGA modes are doubled. Otherwise, the image on the screen would appear in only half its height.

A major disadvantage of the VGA modes on this card is the color selection. Although the MCGA can display the necessary VGA resolution, the card is limited in its color palette because of the small amount of video RAM that's available (only 64K). MCGA cards are so named because they will operate only on Micro Channel systems. This means that only low end PS/2 systems can use the MCGA.

8514/A

Still trying to set video standards, IBM presented a successor to the VGA standard in 1987. This card, ambiguously named the 8514/A video card, caused a revolution in video cards. While earlier video controllers relied upon the main processor for information (i.e., they were "dumb" controllers), this video card had its own processor.

With this feature, graphics functions could be delegated to the graphics processor on the card, instead of requiring the PC's 80x86 processor to calculate these functions. So the graphics are drawn from the video card itself, which frees the PC's processor for other tasks.

So far, the 8514/A hasn't been able to replace the VGA. This may have been caused by poor development and marketing decisions. For instance, this graphic standard was intended for only the PS/2 models and Micro Channel. This immediately reduced the market share. Also, IBM kept the technical details of this card confidential, so third party manufacturers couldn't build compatible copies of the card. This strategy is quite different from IBM's earlier "open system" attitude. Finally, this video card requires a software interface developed by IBM, which developers have avoided. Although this software interface is powerful, sometimes it hinders the hardware's performance. Consequently, the 8514/A has a small following.

The Video BIOS

The PC's ROM BIOS performs many actions for different screen display tasks. These actions are grouped as functions of interrupt 10H (video interrupt functions). Although there are other interrupt 10H functions, we'll discuss only the video BIOS functions. In this section, we'll describe the video BIOS functions, how you can access them and why direct access to video hardware is usually the best method.

The video BIOS and its extensions

Originally, the functions of interrupt 10H applied only to MDA and CGA cards. These functions also support Hercules cards in text mode because the Hercules cards are fully compatible with the MDA standard. The original BIOS doesn't support EGA and VGA cards or their extended features in text and graphics modes. So, EGA and VGA cards include their own BIOS extensions on a ROM chip. These extensions are enabled when you boot the system.

This set of BIOS extensions interact with interrupt 10H to add EGA and VGA functions to the existing BIOS. Although they contain the same extensions, the EGA BIOS has fewer capabilities than the VGA BIOS. EGA and VGA cards are manufactured by several manufacturers, but they have the same BIOS extensions as IBM's EGA and VGA cards. Only a few cards are incompatible with these functions.

Some top-of-the-line PCs are packaged with VGA cards directly on the motherboard. In these instances, the VGA and EGA BIOS functions are added to the ROM BIOS, which eliminates the extensions. However, this doesn't change how the cards are programmed.

Speed and BIOS functions

Using the video BIOS functions isn't the only way to handle tasks such as positioning the cursor or drawing characters on the screen. The DOS screen output functions and any BIOS functions used for direct video hardware programming can also be used. If listed according to effectiveness, the BIOS functions would be located between the DOS functions and direct hardware programming. These functions are used when execution speed, compatibility, device independence and flexibility are important to program development.

The DOS functions offer the most device independence because the output can be sent to the screen, the printer or a disk as a file. However, DOS functions execute slowly and aren't very flexible. Direct hardware programming provides the highest possible execution speed and flexibility because the programmer has absolute control over execution. However, direct hardware programming is extremely hardware dependent. For example, a character output routine written for a CGA won't work when on an MDA.

The BIOS functions aren't as fast as direct access routines, but they will work with the currently installed video card. So, the programmer doesn't have to make a distinction between cards; the BIOS always performs the tasks. You may be wondering why the BIOS functions are slower than direct hardware access. There are two reasons for this. First, the mechanism used to call these BIOS functions is slow. Second, the call to an interrupt takes much longer than a routine within a program. All BIOS routines use this latest technique, called the interrupt programmer.

Many video cards are 8-bit cards, which slow down access to the ROM BIOS. Remember, 80286, 80386 and 80486 processors "think" in 16-bit and 32-bit units. If you consider the PC must execute every assembly language instruction in the BIOS routines as an 8-bit instruction, you can see why the routines are so slow.

With many PCs, you can relocate their ROM BIOS to a range of RAM between video RAM and the 1 Meg memory limit, called shadow ROM. This makes the BIOS routines run considerably faster since the processor can make 16 and 32-bit accesses. However, the disadvantages of direct hardware programming also apply in this instance. While the BIOS functions perform many useful services, most PC applications use a combination of BIOS functions and direct access routines, especially for fast screen output in text and graphics modes. Control tasks, such as video mode initialization, text cursor placement and screen page selection should be handled by the BIOS.

The video BIOS services

Next we'll describe the most important control functions and the services used in text output. We'll discuss other functions in later chapters and in the Appendix. Let's begin with an overview of the different services available from the BIOS video interrupts and their sub-functions.



You'll find the Appendices on the companion CD-ROM

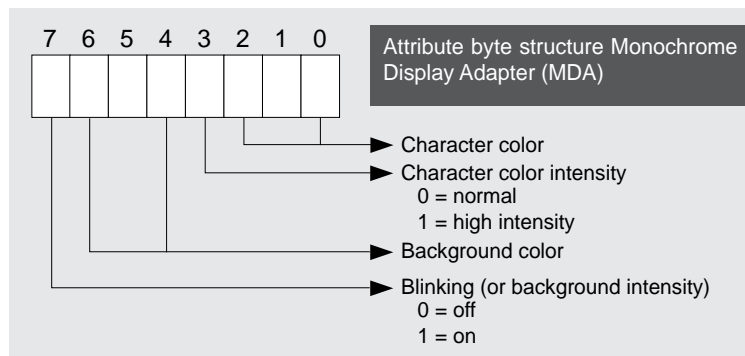
Video BIOS functions and support from EGA, VGA and standard BIOS					
No.	Meaning	BIOS*	No.	Meaning	BIOS*
00H	Determine video mode	SEV	0EH	Terminal character output	SEV
01H	Define cursor size	SEV	0FH	Determine video mode	SEV
02H	Set cursor position	SEV	10H	EGA/VGA color options	EV
03H	Read cursor position	SEV	11H	Character generator access	EV
04H	Read light pen	SEV	12H	Set/read video configuration	EV
05H	Define current screen page	SEV	13H	Write string (AT only)	SEV
06H	Scroll screen up	SEV	14H	Reserved	---
07H	Scroll screen down	SEV	15H	Reserved	---
08H	Read character and attribute	SEV	16H	Reserved	---
09H	Write character and attribute	SEV	17H	Reserved	---
0AH	Write character to cursor position	SEV	18H	Reserved	---
0BH	Set color palette for graphics mode	SEV	19H	Reserved	---
0CH	Set screen pixel in graphics mode	SEV	1BH	Toggle between video cards	V
0DH	Read screen pixel in graphics mode	SEV	1CH	Save/restore video card status	V ;
* S = Standard BIOS E = EGA BIOS V = VGA BIOS					

The first step in calling these functions is to load the AH register with the function number. If a sub-function exists, this number is loaded into the AL register. However, we will mention exceptions to this rule in this book. One exception to the rule are the functions that alter the contents of the registers because most functions usually change registers.

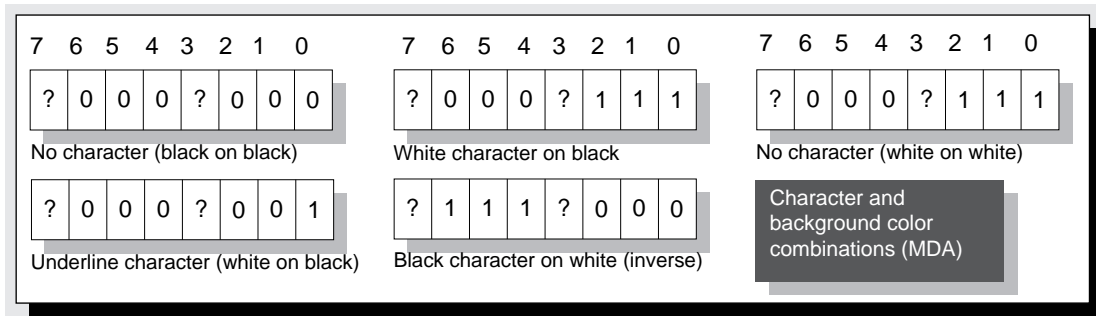
Selecting character and background colors

Some character display functions expect foreground and background colors from the caller. Some video cards have separate systems for setting foreground and background colors.

Unfortunately, monochrome and color video cards determine these colors differently. In both cases, each character has a color or attribute byte divided into two nibbles. The least significant nibble (bits 0-3) defines the foreground color, while the most significant nibble (bits 4-7) defines the background color.

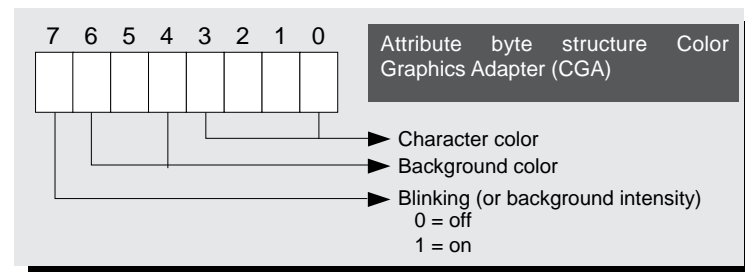


MDA cards divide the nibbles into bits for foreground intensity and character blinking, as you can see in the illustration above. The following illustration shows how the intensity bits for foreground and background colors interact through bit combination:



Frequently selected monochrome video card colors are 07H (light text on dark background) and 70H (dark text on a light background). These codes also work with color cards for color combinations such as light gray on black and black on light gray.

Selecting 0 or 7 as text or background color changes the status of bit 7 in the attribute byte. This byte determines whether the character blinks or the background appears in high intensity color. The illustration above shows the structure of the color attribute bytes.



The color card lets you specify colors for foreground and background from a palette of 16 colors. The following shows the structure of the color palette.

Color/Graphics Adapter color palette							
Decimal	Hex	Bin	Color	Decimal	Hex	Bin	Color
0	00H	0000(b)	Black	8	08H	1000(b)	Dark gray
1	01H	0001(b)	Blue	9	09H	1001(b)	Light blue
2	02H	0010(b)	Green	10	0AH	1010(b)	Light green
3	03H	0011(b)	Cyan	11	0BH	1011(b)	Light cyan
4	04H	0100(b)	Red	12	0CH	1100(b)	Light red
5	05H	0101(b)	Purple	13	0DH	1101(b)	Light purple
6	06H	0110(b)	Brown	14	0EH	1110(b)	Yellow
7	07H	0111(b)	Light gray	15	0FH	1111(b)	White

The ASCII character set

The PC uses a character set based on 256 symbols, numerals, letters and special characters. Many of these special characters are foreign language characters, mathematical symbols and linedrawing characters.

For more information and examples of these characters, refer to the Appendix.



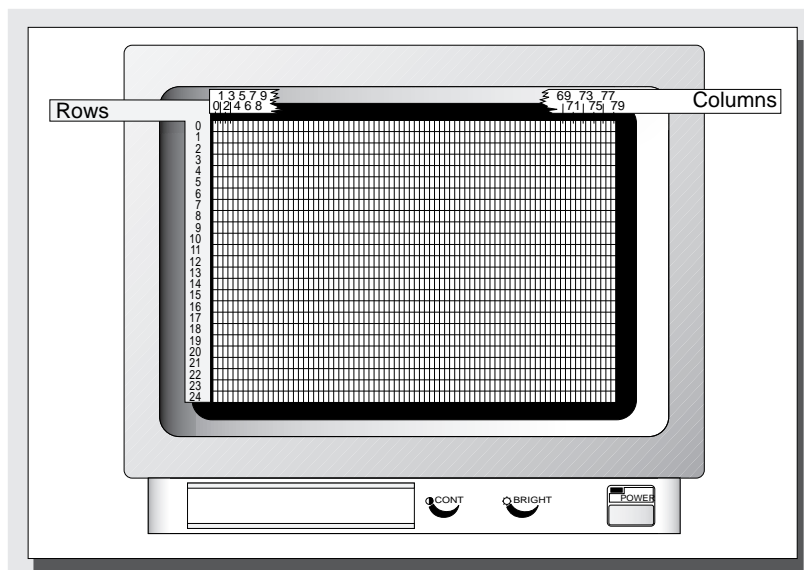
You'll find the Appendices on the companion CD-ROM

The screen coordinate system

Many BIOS functions require screen coordinates as a parameter. These coordinates specify the location on the screen where you want to display the character. You must understand this coordinate system before you can call many of the functions.

Whether in text or graphics mode, the origin of this coordinate system is the upper-left corner of the screen. Moving to the right increments the X-coordinate, while moving down increments the Y-coordinate. In 80x25 character text mode, the lower-right screen corner is coordinate 79/24, while the lower-right corner of a CGA card's 640x200 pixel graphics mode is coordinate 639/199.

Screen row and column numbering



Initializing a video mode

Using function 00H initializes the video mode of a graphics card. Placing 00H in the AH register and a sub-function code in the AL register initializes the standard video mode in text or graphics mode (except graphics mode on the Hercules card).

Initializing a video mode assumes the corresponding video card is installed. If you initialize a video card or mode that doesn't exist, the system may crash. When you call function 00H, the contents of video RAM are cleared and the selected video mode is initialized. The contents can be retained on EGA and VGA cards by adding 128 to the mode number (i.e., by setting bit 7 in the mode number). Calling function 00H in this way keeps the contents of video RAM intact and displays these contents on the screen after initialization.

You can immediately set 80x25 character text mode as active when a program starts. This is mode 7 on MDAs and mode 3 on CGAs. You don't need to call function 00H when you want your program to operate in 80x25 text mode. Function 0FH reads the current video mode. Call this function by passing 0FH in the AH register. After you call the function, the AL register returns a value. Use the table previously listed to determine the currently active video mode. The number of columns per screen line is returned in the AH register (if this mode is a text mode). The number of current screen pages, if applicable, is returned in the BH register.

Video mode sub-functions from video BIOS function 00H		
Code	Mode	Card
00H	40x25 character text, 16 colors, no color display	CEV
01H	40x25 character text, 16 colors	CEV
02H	80x25 character text, 16 colors, no color display	CEV
03H	80x25 character text, 16 colors	CEV
04H	320x200 pixel graphics, 4 colors	CEV
05H	320x200 pixel graphics, 4 colors, no color display	CEV
06H	640x200 pixel graphics, 2 colors	CEV
07H	80x25 character text, mono	MHE*
08H	Reserved	
0CH	Reserved	
0DH	320x200 pixel graphics, 16 colors	EV
0EH	640x200 pixel graphics, 16 colors	EV
0FH	640x350 pixel graphics, mono	E*
10H	640x350 pixel graphics, 16 colors	EV
11H	640x480 pixel graphics, 2 colors	V
12H	640x480 pixel graphics, 16 colors	V
13H	320x200 pixel graphics, 256 colors	V
EGA card on MDA monitor M = MDA H = Hercules C = CGA E = EGA V = VGA		

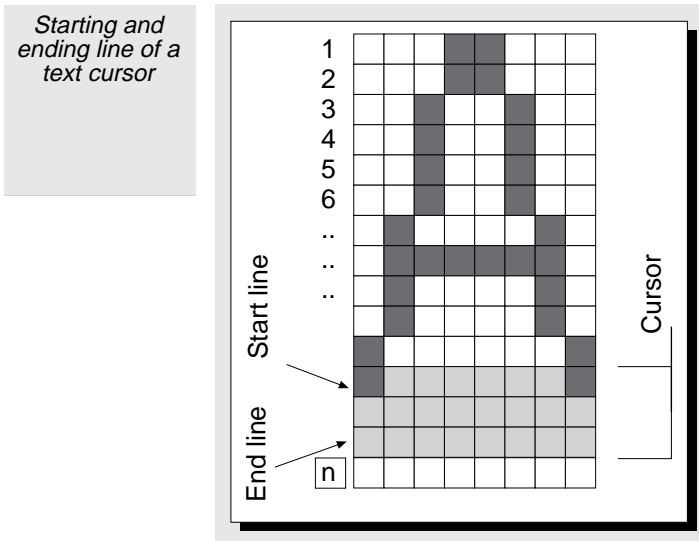
Programming the text cursor

In text mode, every video card from MDA to VGA has a blinking cursor. This cursor indicates the current input or output position. The video BIOS controls both the appearance and screen position of this cursor.

Function 02H handles cursor positioning. Place the function number (02H) in the AH register. Place the row where you want to locate the cursor in the DH register and place the column where you want to locate the cursor in the DL register. Also, place the number of the screen page, at which you want the cursor located, in the BH register. This is applicable only if each page has its own cursor available. The blinking cursor only appears when the value in the BH register corresponds to the current screen page. This function call determines the next location at which screen input and output will occur. Refer to the Appendix for more information about this function.

Function 03H reads the current cursor position in a specified screen page and returns this position to the program that called the function. Place the function number (03H) in the AL register and the screen page that should be read in the BH register. This function returns the cursor position in the CH register (starting pixel line of the cursor) and the CL register (ending pixel line of the cursor), instead of the actual position.

To understand these values, remember that a character in text mode on a color card is eight pixels high and that a character in text mode on a monochrome card is 14 pixels high (not screen rows). These values tell the programmer where the blinking cursor begins and ends.



These values also provide information about the height of the character matrix, from which you can determine the sizes of the characters. Since the CGA card generates characters that are eight pixels high, the starting and ending lines should be from 0 to 7. Since a Hercules and an MDA card generate characters that are 14 pixels high, the cursor values range from 0 to 13. The EGA and VGA cards use even higher values, but the CGA measurement of 0 to 7 is used here. The actual character matrix can be recalculated from these values.

Greater values for the starting and ending lines can occur when the cursor disappears from the screen.

Function 01H defines the appearance of the cursor. To do this, place 1 in the AH register, the starting line in the CH register and the ending line in the CL register. Be sure the

starting line is less than or equal to the ending line; otherwise the cursor will no longer be visible.

Selecting the screen page

Although we've mentioned the current screen page, we haven't explained how to activate a screen page. Function 05H of the video BIOS performs this task. Place the value 05H in the AH register and the number of the screen page you want activated in the AL register. The screen page number will vary depending on the number of pages available in the video card. For example, since the MDA has only one page, calling this function for an MDA card is useless.

The table on the right shows values that apply to the video cards that support multiple screen pages with their video modes. Screen page numbering always begins at 0. So, an EGA or VGA card in mode 2 can access screen pages 0 to 7.

Number of available screen pages depends on video card and video mode			
Mode	Resolution	Card	Pages
7	80x25	MDA/Hercules	1
0/1	40x25	CGA	8
2/3	80x25	CGA	4
0/1	40x25	EGA/VGA	16
2/3	80x25	EGA/VGA	8

Character output and BIOS

The video BIOS contains various character output functions. Each function handles control codes differently. These control codes consist of ASCII codes 7, 8, 10 and 13. Although the IBM system views them as normal characters, data processing history considers these characters text controls (see the table on the right).

Some functions view these codes as normal ASCII characters and display them as such. Other functions execute the controls specified by these codes. For example, code 7 instructs the computer to sound a beep. The function you select determines the actions performed by these codes.

PC ASCII control characters		
ASCII code	Name	Purpose
7	Bell	Sounds beep
8	Backspace	Deletes character left of cursor and moves cursor right one character
10	Linefeed	Moves cursor to next line
13	Carriage Return	Moves cursor to beginning of current line

Remember, all text output functions operate in both text mode and graphics mode. Character output in graphics mode isn't directly accessible because a character set isn't available. However, BIOS compensates for this limitation by setting the ASCII character patterns as graphic pixels. While the character patterns for ASCII codes 0 to 127 are already stored in ROM, codes 128 to 255 are taken from a table in RAM, which is installed by the GRAFTABL command from MS-DOS.

BIOS removes the address of this table as a FAR pointer (you'll find the table starting at 0000:007C). Although these memory addresses lie within the interrupt vector table, interrupt 1FH, which normally uses this address, cannot be used.

The condition that stores this table in RAM enables you to design your own table. With a user-defined table, special characters, which aren't found in the standard character table, can be displayed on the screen. Each character requires eight bytes. The first eight bytes in the table define ASCII code 128, the second eight bytes define ASCII code 129, etc. Each byte represents the bit pattern for one of the eight lines used in each character. Bit 0 represents the right border of each character matrix, while bit 7 represents the left border of each character matrix. If a bit is set as 1, the corresponding pixel appears on the screen.

Although functions 09H and 0AH both display characters, there is a difference between them. Function 0AH displays the character in the color established for that position on the screen and function 09H displays the color (the attribute) set by the character itself. After character output, both functions keep the cursor at the same cursor position so the next call of either function places character output at that same location.

Function 02H moves the cursor to the next screen position.

Both functions interpret control codes as normal characters and display these characters as such. Place the function number in the AH register and the ASCII code you want displayed in the AL register. The BH register contains the screen page, on which the character should be displayed (where applicable). The CX register contains a number that indicates how often the output should follow. This enables you to display a single character several times in one function call. If the character in the AL register should be displayed only once, the CX register should contain the value 1.

Because of an error in BIOS, the repeat factor during the call of this function in graphics mode should be limited to the maximum number of characters that can be displayed in one line.

Function 09H passes the character and its color. Place the character color number in the BL register.

Both functions have a disadvantage. The cursor remains at the same cursor position, unlike function 0EH, which increments the cursor to the next screen position. It simulates a terminal; this process is often referred to as the TTY routine (teletype routine) routine. Calling function 0EH displays the character and increments the cursor to the next character. If the cursor reaches the end of a screen line, the cursor jumps to the beginning of the next screen line.

If the cursor reaches the lower-right corner of the screen (column 79, line 24), the entire contents of the screen scroll up one line and the cursor moves to the first column of line 24.

Unlike functions 09H and 0AH, the TTY function handles the control codes as control codes instead of as normal ASCII characters. The TTY function displays the character in the color previously defined for that screen location. This function applies only to text mode. In graphics mode, the TTY function must have the character color stored in the BL register.

Place the function number 0EH in the AH register, the code you want displayed in the AL register and the screen page, in which the character should be displayed, in the BH register.

String output

When the AT was introduced, the video BIOS included a new function (13H), which was also found in the BIOS versions included on EGA and VGA cards. This function displays a character string on the screen with a single function call.

Place the function number in the AH register and the screen page to be displayed in the BH register. Place the starting position of the string in the DH register (row) and the DL register (column). The CX register should contain the number of characters to be displayed in the string.

The AL register contents define one of four available modes, in which the string can be displayed. Modes 0 and 1 specify the format in the first screen page and modes 2 and 3 specify the format in other screen pages. Modes 0 and 1 contain only the characters to be displayed, but modes 2 and 3 include both characters and attribute bytes for each character. The BL registers should contain the attribute bytes for all characters. Whatever buffers are allocated, the ES:BP register pair must contain a FAR pointer to the buffer.

Modes 2 and 3 contain two bytes (character byte and attribute byte) for every character in the string. So, a string that's four characters long actually contains eight characters. However, the CX register should contain the number 4 (i.e., the number of bytes in the actual string). There's another difference between modes 0 and 2 and modes 1 and 3. After screen output in modes 1 and 3, the cursor moves to the next screen position so BIOS output continues at this point. However, in modes 0 and 2, the cursor position isn't updated.

Reading characters from the screen

While functions 09H, 0AH and 0EH display characters, function 08H reads characters currently on the screen. The function senses which character is at a particular screen position and which attribute applies to that character. Place the function number in the AH register and the screen page number in the BH register. The screen position is the current cursor position.

Although the character code can be read directly from video RAM in text mode, in graphics mode, the character pattern at the current cursor position must be compared with all available character patterns. However, since this doesn't always work, you shouldn't rely on this function in graphics mode.

The function returns the attribute (color) in the AH register and returns the ASCII code of the character in the AL register.

Screen scrolling

In the description of the TTY output function (0EH) we mentioned the screen scrolls (moves up) when the cursor reaches the last column and line on the screen.

Function 0EH executes an internal call to function 06H to perform the scrolling. Function 06H scrolls the screen area one or more lines up, displaying a blank space at the bottom of the screen. Only the currently displayed screen page is affected by this operation. Place the function number 06H in the AH register and the number of lines you want scrolled in the AL register. If you place a value of 0 in this register, instead of being scrolled, the line(s) will be filled with spaces. In this case, the BH register contains the color you want assigned to the blank line(s). The CH, CL, DH and DL registers define the screen range.

CH	Upper-left window corner (line)
CL	Upper-left window corner (column)
DH	Lower-right window corner (line)
DL	Lower-right window corner (column)

Function 07H scrolls the screen window down rather than up. The same parameters are used for function 07H and function 06H.

Sample programs

The following programs demonstrate how to use the BIOS video interrupt functions that are available from higher level languages. In Pascal and C, you'll find that using BIOS display functions works much faster than the standard procedures and functions, which use the slower DOS functions, that are included in these languages.

Advantage

An advantage of accessing BIOS video interrupt functions instead of using onboard graphics commands in higher level languages is the BIOS function can be accessed at any time.

Disadvantage

However, there is a disadvantage to using BIOS functions for screen output. The higher level language display commands can accept numeric variables, which are then converted to ASCII characters. These higher level commands can format the variables according to decimal places (or a certain degree of precision) and then display them. However, if numeric variables are displayed using the BIOS functions, first they must be converted into a character string that must be transferred to the BIOS output function. Obviously, this procedure is very time-consuming.

Both programs operate the same way. Each fills the screen with continuous characters from the PC character set, then opens two windows in which two arrows move up and down. You'll understand how this was done and how it will actually appear on the screen, after you've studied the program codes. The programs limit their access to one screen page because of incompatibility problems that could occur between monochrome and color cards. Also, they don't present subroutines, functions or procedures for calling the BIOS graphics functions.

Once you understand this section you should be able to add the missing functions and even write a short demonstration program of your own. Using the BIOS video interrupt ensures the computer won't crash and that you won't encounter serious problems.

The individual functions and procedures of the VIDEO.PAS and VIDEOC.C programs which you'll find on the companion CD-ROM are fully documented and should be self-explanatory. These programs look similar because the procedures, functions and variables have the same names.

You'll find the following program(s) on the companion CD-ROM



VIDEO.PAS (Pascal listing)
VIDEOC.C (C listing)

Determining Video Card Type

Whenever you want to access the video card hardware or use a BIOS function that's only available in special versions of the BIOS, first you must ensure the card is actually installed in the system. If you don't do this, the image that appears on the screen may look different than what you expected.

If the program is supposed to be compatible with all types of cards, it's especially important that it recognize the type of video card that's installed, while still directly accessing video hardware. The output routines need this information to use the special properties of the given card effectively.

Remember the PC can have both a monochrome video card (MDA, HGC or EGA with a monochrome monitor) and a color video card (EGA, VGA or CGA) installed. However, only one of the cards can be active at a time.

Combinations of PC video cards					
	VGA	EGA	HGC	CGA	MDA
VGA			X		X
EGA			X	X	
HGC	X	X		X	
CGA		X	X		X
MDA	X	X		X	

We must determine which video cards are installed. BIOS or DOS functions cannot be used to do this and variables cannot be read. So, we must write an assembly language routine that checks for the existence of different video cards.

Refer to the documentation for the various cards. Manufacturers usually include a procedure for determining whether their card is being used. It's important to keep the test specific (i.e., it doesn't return a positive result if a certain type of video card isn't installed). This is difficult to do with EGA and VGA cards, which can emulate CGA or MDA cards with the appropriate monitor and are difficult to distinguish from true CGA or MDA cards.

However, not all registers can be described and read in this case. Registers 14 and 15, which determine the address of the blinking cursor, don't cause any problems.

All the tests we present are located at the end of this section in the form of two assembly language programs. These programs are intended to be used with C and Pascal programs. The functions place the type of video card installed and the type of monitor connected to it into an array. The function is passed a pointer to this array. If two video cards are installed, their order in the array indicates which one is active.

The following cards can be detected by the assembly language routine:

- MDA cards ➤ CGA cards ➤ HGC cards
- EGA cards ➤ VGA cards

Since the assembly language routine checks for the existence of a specific video card, there is a separate subroutine for each type of card. The name of the subroutine is the same as the video card for which it tests. For example, these routines could be called TEST_EGA, TEST_VGA, etc.

Although these tests can be called sequentially, certain tests can be excluded if you know they would return a negative result. For example, this applies to the CGA test if an EGA or VGA card has already been detected and is connected to a high resolution color monitor. Since a CGA card cannot be installed along with this type of card, you don't need to test for it.

A flag for each test determines whether the test will be performed. Before the first test (the VGA test), all the flags are set to 1 so all the tests will be performed sequentially. During the testing, certain flags can be set to 0 so the corresponding tests won't be performed.

VGA test

The tests begin with the VGA test. A special function in the VGA BIOS, sub-function 00H of function 1AH, returns the information needed by the assembly language routine. The information is available only if a VGA card and a VGA BIOS are installed. This is the case if the value 1AH is found in the AL register after the call. If the test routine encounters a different value there, the VGA test will be terminated and the other tests will be performed. This indicates that a VGA card is not installed. Based on this information, the sequence of both entries is exchanged in the last step of the assembler routine.

After this function is called, the BL register contains a special device code for the active video card and the BH register contains a code for the inactive card. The table on the left shows which errors can occur.

Return Code of Function 00H of Function 1AH	
Code	Meaning
00H	No video card
01H	MDA card/monochrome monitor
02H	CGA card/color monitor
03H	Reserved
04H	EGA card/high resolution monitor
05H	EGA card/monochrome monitor
06H	Reserved
07H	VGA card/analog monochrome monitor
08H	VGA card/analog color monitor

These codes are separated into values for the video card and the monitor connected to it and loaded into the array whose address is passed to the assembly language routine. Since this routine already has information about both video cards, the following tests don't have to be performed. However, the routine executes the monochrome test if the functions discover a monochrome card, because it cannot distinguish between an MDA and HGC card.

EGA test

After the VGA test, the EGA test is performed only if the VGA test was unsuccessful and, consequently, the EGA flag wasn't cleared. This test uses a function that's found only in the EGA BIOS, called sub-function 10H of function 12H. If an EGA card isn't installed and this function isn't available, the value 10H will still be found in the BL register after the function call. In this case, the EGA test ends.

If an EGA card is installed, the CL register will contain the settings of the DIP switches on the EGA card after the call. These switches indicate which type of monitor is connected. They are converted to the monitor codes the assembly language routine uses and placed in the array along with the code for the EGA card. The CGA or monochrome test flag is cleared, depending on the type of monitor connected. The EGA routine ends.

CGA test

If the CGA flag hasn't been cleared by the previous tests, the CGA test follows the EGA test. As with the monochrome test, there aren't any special BIOS functions that can be used and we must check for the presence of the appropriate hardware. In both routines this is done by calling the routine TEST_6845, which tests to determine whether the 6845 video controller found on these cards is at the specified port address. On a CGA card this is port address 3D4H, which is passed to the routine TEST_6845.

The only way to test the existence of the CRTC at a given port address is to write some value (other than 0) to one of the CRTC registers and then read it back immediately. If the value read matches the value written, then the CRTC and, therefore, the video card are present.

Before writing a value into a CRTC register, you should remember these registers have a major impact on the makeup of the video signals. So, carelessly accessing them can not only thoroughly confuse the CRTC, but also harm the monitor. Registers 00H to 09H cannot be used for this test, so only registers 0AH to 0FH are available. All these registers affect the screen contents. We can use registers 0AH and 0BH, which control the starting and ending lines of the cursor.

The assembly language routine first reads the contents of register 0AH before it loads any value into this register. After a short pause so the CRTC can react to the output, the contents of this register are read back. Before the value read is compared to the original value, the old value is first written back into the register so the screen is disturbed as little as possible. If the comparison is positive, then a CRTC is present and so is the video card (CGA in this case). The CGA routine responds by loading the code for a color monitor into the array because this is the only type of monitor that can be used with a CGA card.

Monochrome test

The last test is the monochrome test, which also checks for the existence of a CRTC; except this time port address 3B4H is checked. If it finds a CRTC there, then a monochrome card is installed and we must determine whether it's an MDA or HGC card. The status registers of the two cards, at port address 3BAH, are used to determine this. Since bit 7 of this register is meaningless on the MDA card, its value is undefined. However, it contains a 1 on an HGC card whenever the electron beam is returning across the screen. Since this isn't permanent and occurs only at intervals of about two milliseconds, the contents of this bit constantly alternate between 0 and 1.

Hercules

The test routine first reads the contents of this register and masks out bits 0 to 6. The resulting value is used in a maximum of 32,768 loop passes, in which the value is read again and compared with the original value. If the value changes, which means the state of bit 7 changes, then an HGC card is probably installed. If this bit doesn't change over the course of 32768 loop passes, then an MDA card is being used.

Again, we place the appropriate code for the video card in the array. The monitor code is also set to monochrome, since this is the only monitor that can be connected to an MDA card or an HGC card.

Primary and secondary video systems

The tests are now complete. Next we must determine which card is active (primary) and which is inactive (secondary). If the outcome of the VGA test was positive, we can skip this because the VGA BIOS routine determines the active card automatically.

In other instances, we can determine the active video card from the current video mode, which can be read with the help of function 0FH of the BIOS video interrupt. If the value seven is returned, then the 80x25 text mode of the monochrome card is active. All the other modes indicate that a CGA, EGA or VGA card is active. This information is used to exchange the order of the two entries in the array if it doesn't match the actual situation.

The assembly language routine returns control to the calling program.

We've included C and Pascal programs that call the function GetVIOS from the assembly language module and demonstrate how GetVIOS works. You'll find the source code and programs and modules on the companion CD-ROM.

You'll find the following program(s) on the companion CD-ROM



VIOSP.PAS (Pascal listing)
VIOSPA.ASM (Assembler listing)
VIOSPC.C (C listing)

Anatomy Of A Video Card

Regardless of whether a card is an old MDA or the newest Super VGA card, all video cards operate under the same principles. Before we discuss each video standard in detail, we'll look at the general design of video cards. You'll learn how a monitor generates a video image and how to control the CRT controller. We'll also discuss the CRT controller's registers.

This section also contains information about video RAM and how you can use it for creating a video image. In addition, you'll learn the basics of video RAM programming.

Getting to the screen

The character generator first accesses video RAM and reads each characters individually. It uses a character pattern table to construct the bitmap that will later form the character on the screen. The attribute controller also receives information about the display attributes (color, underlining, reverse, etc.) of the character from the video RAM.

Both the character generator and the attribute controller prepare this information and send it to the signal controller. This controller converts the information to the appropriate signals, which are sent to the monitor. The signal controller itself is controlled by the CRT controller, which is the central point of video card operations. Besides the monitor and the video RAM, this CRT controller is one of the most important components of a video system. We'll discuss all these components in detail.

The monitor

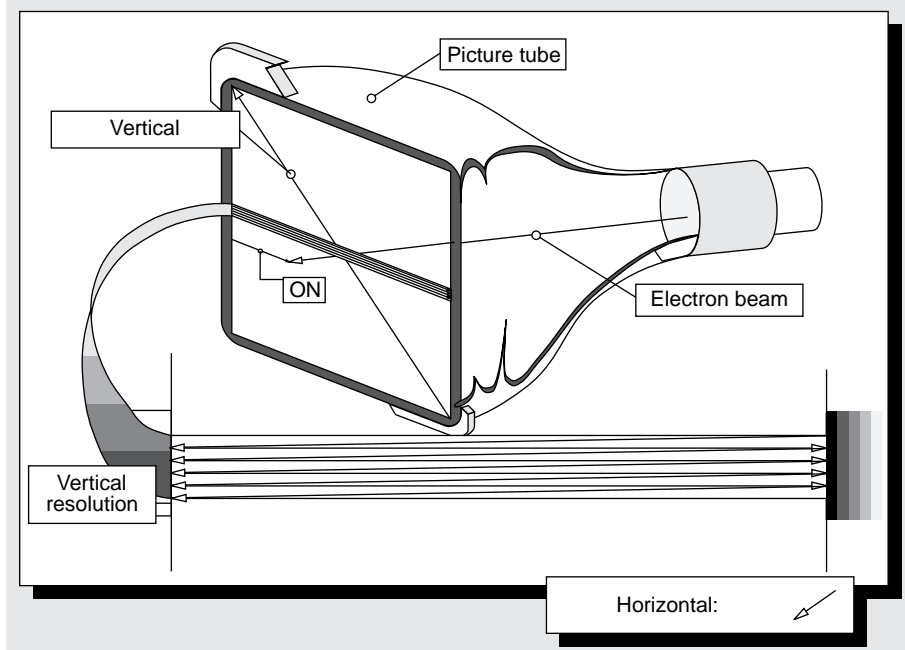
The monitor is the device on which the video data is displayed. Unlike the video card, the monitor is a "dumb" device. This means that it has no memory and cannot be programmed. All monitors used with PCs are raster-scan devices, in which the picture consists of many small dots arranged in a rectangular pattern or raster.

When forming the picture, the electron beam of the picture tube touches each individual dot and illuminates it if it's supposed to be visible on the screen. This is done by switching on the electron beam as it passes over this dot, which causes a phosphor particle on the picture tube to light up.

Color monitors

While monochrome monitors need only one electron beam to create a picture, color monitors; use three beams that scan the screen simultaneously. Here a screen pixel consists of three phosphor particles in the basic colors of light: Red, green and blue. Each color has a matching electron beam. Any color in the spectrum can be created by combining these three colors and varying their intensities.

Electron beam scan movement



However, since an ionized phosphor particle emits light for only a very brief period of time, the entire screen must be scanned many times per second to create the illusion of a stationary picture. PC monitors perform this task 50 to 70 times per second. This repeated re-scanning is called the *refresh rate*. Generally, the quality of the picture improves as the refresh rate increases.

Each new screen image begins in the upper-left corner of the screen. From there, the electron beam moves to the right along the first raster line. When it reaches the end of this line, the electron beam moves back to the start of the next line down, similar to pressing the **Return** key on a typewriter. The electron beam then scans the second raster line. At the end of this line it moves to the start of the next raster line, etc. Once it reaches the bottom of the screen, the electron beam returns to the upper-left corner of the screen and the process restarts. The illustration above shows the path of the electron beam.

Remember the video card and not the monitor, controls the movement of the electron beam. The resolution of the monitor naturally controls the number of raster lines and columns, which the electron beam scans when creating a display. So, a monitor that has only 200 raster lines of 640 raster columns each cannot handle the high resolutions of an EGA card at 640x350 pixels. The table on the left shows the resolutions on different monitor types used with a PC.

The CRT controller

The CRT controller or CRTC, is the heart of a video card. It controls the operation of the video card and generates the signals the monitor needs to create the image. Its tasks also include controlling light pens, generating the cursor and controlling the video RAM. To inform the monitor of the next raster line, the CRTC sends a display enable signal at the start of each line. This signal activates the electron beam. While the beam moves from left to right over each raster column of the line, the CRTC controls the individual signals for the electron beam(s) so the pixels appear on the screen as desired. At the end of the line, the CRTC disables the display enable signal so the electron beam's return to the next raster line doesn't produce a visible line.

Resolution on different monitor types

Monitor	Raster lines	Raster columns
MDA/Hercules	350	720
CGA	200	640
EGA	350	640
VGA*	350	640
Super VGA*	600	800
Multisync	Varies to 800	Varies to 1200

*Available in monochrome and color versions

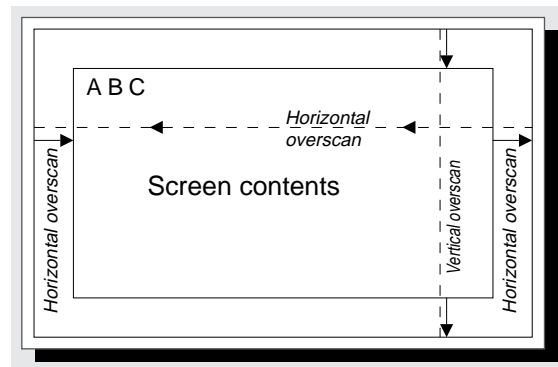
on the screen. The electron beam is directed to the left edge of the following raster line by the output of a horizontal synchronization signal. The display enable signal is again enabled at the start of the next raster line and the generation of the next line begins.

Overscan

Since the time the electron beam needs to return to the start of the next line is less than the time the CRTC needs to receive and prepare new information from the video RAM, there is a short pause. However, the electron beam cannot be stopped, so overscan occurs, which is visible as the left and right borders of the actual screen contents. Although this is an undesirable side effect, it's useful because it prevents the edges of the screen contents from being hidden by the edge of the monitor. If the electron beam is enabled while it's traveling over this border, a color screen border can be created.

Once the electron beam reaches the end of the last raster line, the display enable signal is disabled and a vertical synchronization signal is sent. The electron beam returns to the upper-left corner of the screen. Again the display enable signal is re-enabled and scanning begins again.

Rasters and overscan on a screen



As with the horizontal electron beam return, a pause occurs. This pause is displayed in the form of overscan, which creates a vertical screen border.

The registers of the CRT controller

The timing of individual signals varies depending on the video mode. Therefore, the CRTC has numerous registers that describe the signal outputs and their timing. The structure of these registers and how they are programmed will be discussed in the remainder of this section. Many of these registers originate from the registers of the 6845 video controller from Motorola. This controller is used in the MDA, CGA and Hercules graphics cards. The EGA and VGA cards use a special VLSI (very large scale integration) chip as a CRTC and its registers are slightly more complicated. The techniques described here are intended as general descriptions for all video cards.

These registers, like all the other registers on the video card, are accessed using I/O ports with the assembly language instructions IN and OUT. Instead of being accessed directly from the address space

Motorola 6845 video controller registers		
Register	Meaning	Access
0CH	Starting address of screen page (high byte)	Write
0DH	Starting address of screen page (low byte)	Write
0EH	Character address of blinking screen cursor (high byte)	Read /Write
0FH	Character address of blinking screen cursor (low byte)	Read /Write
10H	Light pen position (high byte)	Read
11H	Light pen position (low byte)	Read

of the processor, the registers of the CRTC are accessed through a special address register. The number of the desired CRTC register is written to the port corresponding to this address register. Then the contents of this register can be read into a special data register with the IN assembly language instruction. If a value must be written to the addressed register, it must be transferred to the data register with the OUT instruction. Then the CRTC automatically places it in the desired register. Although these two registers are actually found at successive port addresses, these addresses vary among video cards.

Monochrome video card port addresses are 3B4H and 3B5H and color card port addresses are 3D4H and 3D5H. We've used tables throughout this chapter to show the contents of individual CRTC registers under the various video modes. The following example shows how the contents of these registers are calculated and how the individual registers are related to

each other. If you try some of these calculations with your calculator or PC, you'll notice that some of them do not work out evenly. But since the registers of the CRTC hold only integer values, they will be rounded up or down.

The basis for the various calculations are the bandwidth and the horizontal and vertical scan rates of a monitor. As the following table shows, MDA, CG and Hercules cards operate with a single bandwidth, while EGA and VGA cards have two or more different bandwidths. Although Super VGA cards use normal VGA bands, they can operate with more bandwidths.

Bandwidths and scan rates of different video cards				
Video card	Resolution	Bandwidth	Vertical scan rate	Horizontal scan rate
MDA	720 x 350	16.257 MHz	50 Hz	18.43 KHz
CGA	640 x 200	14.318 MHz	60 Hz	15.75 KHz
HGC	640 x 200	14.318 MHz	50 Hz	18.43 KHz
EGA	640 x 350	16.257 MHz	60 Hz	21.85 KHz
	640 x 200	14.318 MHz	60 Hz	15.75 KHz
	720 x 350	16.257 MHz	50 Hz	18.43 KHz
VGA	640 x 350	16.257 MHz	60 Hz	21.85 KHz
	640 x 200	14.318 MHz	60 Hz	15.75 KHz
	720 x 480	28.000 MHz	70 Hz	31.50 KHz

The bandwidths in the table on the left specify the number of points the electron beam scans per second. This is also called the *point rate* or *dot rate*. The *vertical scan rate* specifies the number of screen refreshes per second, while the *horizontal scan rate* refers to the number of raster lines the electron beam scans per second.

Starting with these values, let's practice calculating the individual CRTC register values for the 80x25 character text mode on a CGA card. You can occasionally determine the number of bands supported by a video card by looking at the circuitry of the card. The video card needs a quartz crystal for each bandwidth and for keeping the CRT controller as accurate as possible. You can easily identify the crystal components on the board by looking for a component or set of components that are silver instead of black. The timing rate should be printed on these crystals.

If you want to look at the Hercules card's graphics mode, you can activate it by directly accessing the different CRTC registers. Function 00H initializes these registers. However, it's interesting to look at the different values that apply to each CRTC register, such as the values needed to produce 80x25 character text mode on a CGA card.

Since a CGA card has an 8x8 character matrix, a screen resolution of 80x25 characters corresponds to a "graphic" resolution of 640x200 pixels. This mode amounts to a bandwidth of 14.318 MHz, a screen refresh frequency of 60 Hz and a horizontal scan rate of 15.740 KHz.

Bandwidth	14.318 MHz
÷ Horizontal scan rate	15.570 KHz
<hr/>	
Pixels per line	919

Raster lines	262
÷ Pixels per character	8
<hr/>	
Lines per screen	32

To obtain the number of pixels (screen dots) per raster line, divide the bandwidth by the horizontal scan rate. Since the CRTC registers usually refer to the number of characters instead of pixels, this value must be converted to the number of characters per line. This is done by dividing the number of pixels per line by the width of the character matrix (see illustration on the left). This is eight pixels on the CGA card.

This value, decremented by one, is placed in the first register of the CRTC and specifies the total number of characters per line. In the second register we load the number of characters that will actually be displayed per line. The 80x25 character text mode usually provides 80 characters.

The difference between the total and the number of characters actually displayed per line is the number of characters that can be displayed between the horizontal return and the overscan. In this instance, the difference is 34 characters.

The duration of the horizontal beam return must be entered in the fourth register of the CRTC. This register stores the number of characters that could be displayed during this time, rather than the actual time duration. The monitor specifications define this instead of the video card itself. Usually this number is between 5% and 15% of the total number of characters per line. A color monitor uses exactly ten characters.

This leaves 24 characters for the overscan (the horizontal screen border). The third CRTC register specifies how these characters are divided between the left and right screen borders. This register specifies the number of character positions that will be scanned before the horizontal beam return occurs. The BIOS specifies the value 90 here or after ten characters have been displayed for the screen borders. The remaining 14 characters are placed at the start of the next line and form the left screen border.

The calculations for the vertical data, the number of vertical lines, the position of the vertical synchronization signal, etc., follow a similar scheme. The first calculation is the number of raster lines per screen. This results from the division of the number of lines displayed per second by the number of screen refreshes per second (see illustration on the right):

Since the characters in CGA text mode are eight pixels high by eight pixels wide, again we divide by eight to obtain the number of text lines per screen.

This result must be decremented by 1 and loaded into the fifth CRTC register (resulting in 7). The seventh register receives the number of lines to be displayed per screen (25). Seven lines less are displayed on the screen. So after overscan, the vertical rescan occurs after the 28th line.

The character height must be decremented by one and loaded into CRTC register 9. The decrement result is 7 in this example. This value also determines the range for the values loaded into registers 0AH and 0BH. They specify the first and last raster lines of the screen cursor. The cursor position is determined by the contents of registers 0EH and 0FH. They refer to the distance of the character from the upper-left corner of the screen, instead of line and column. This value is calculated by multiplying the cursor line by the number of columns per line and then adding the cursor column. The high byte of the result must be loaded into register 0EH and the low byte in register 0FH.

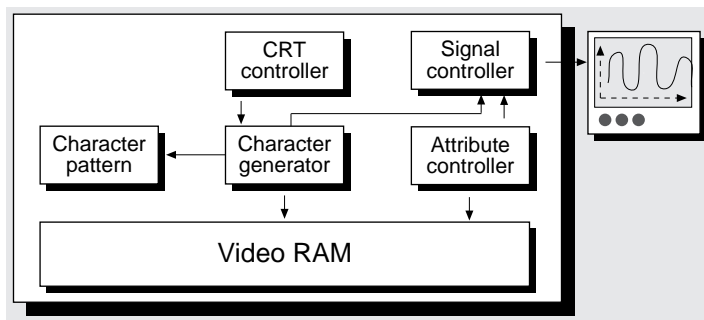
Pixels per line	919
÷ Pixels per character	8
<hr/>	
Characters per line	114
Horizontal scan rate	15.750 KHz
÷ Screen refreshes	60 Hz
<hr/>	
Raster lines	262

Raster lines	262
÷ Pixels per character	8
<hr/>	
Lines per screen	32

Structure of a video card

The CRT controller provides many tasks for screen design, but these tasks cannot work on their own. A video card consists of additional function levels which interact with the CRT controller. The following illustration shows the block diagram of a video card.

*Block diagram
of a video card*



The video RAM is the most important starting point for creating a graphic. Video RAM is a memory range directly connected to the video card (more on this later). When the video card operates in text mode, video RAM contains the ASCII character codes and the character colors, while in graphic mode, video RAM is used to store individual pixels and the number of bits needed for colors.

Video RAM accessed the character generator and attribute controller from text mode. First, an instruction from the CRT controller loads a character from video RAM and generates the character from pixels. A ROM chip containing all the character patterns found in the ASCII character set converts the patterns into table form. The result of this conversion is then passed directly to the signal controller, which is driven by the CRT controller. This signal controller instructs the lines in the monitor cable to display the character pattern.

The signal controller and attribute controller prepare for output. In text mode, the CRT controller is responsible for conveying the character color and reading the signal controller. Although other orders aren't needed for graphics mode, the alternate structure of video RAM must be remembered.

If you look at an early video card, you'll see that it's easy to visually divide it into function units. One reason this is possible is because of the different ICs scattered around the card. Because of advances in miniaturization, almost all components in modern video cards can be packed into a single IC. This applies to EGA and VGA cards, which have a more complicated design than older cards, but still closely match the previous block diagram.

In the following sections we'll present detailed information about the design of specific video cards.

Video RAM

Video RAM is a location common to all video cards (from MDA to Super VGA), whether in text mode or graphics mode. Before you can directly program a graphics card (i.e., communicate with the card's ROM BIOS), you must know the design and position of video RAM. While video RAM is organized differently in different video cards when in graphics mode, the structure is virtually identical for all video cards in text mode.

In this section you'll learn about video RAM organization in different graphics modes. You'll find descriptions of different video cards, as well as how to set and read different graphics modes.

Video RAM in the PC's address space

You may already know that a PC's RAM doesn't always have to be on the motherboard: It can also exist on an expansion card. Video cards often use this expansion RAM as address space and video RAM. Although the video cards themselves are established, the video RAM found in a PC's address range isn't completely flexible. Much of this space is already committed to other tasks. Remember the early PCs had to store RAM, ROM and any system extensions into one megabyte. Video cards only need memory segments A and B, starting at segment addresses A000H and B000H, which requires only 64K.

For the user to use these features, both a monochrome card and color card must be running simultaneously. The B segment would be divided into two equal parts. Monochrome video cards (i.e., MDA and Hercules) use video RAM in the range B000:0000 to B000:7FFF. Color cards (i.e., CGA, EGA and VGA) use video RAM starting at B800:0000.

The locations of video RAM in EGA and VGA cards isn't always clear. For example, an MDA monitor can be connected to an EGA card and the EGA card will then emulate an MDA card. Then video RAM begins at B000H (which would also apply to an MDA) instead of B800H. The case is similar with VGA cards. You cannot connect an MDA monitor to a VGA card, but you can run a VGA monitor from the MDA. In either case, the video RAM begins at B000H.

However, a video card won't always use the entire 32K that's allocated for it. So, the MDA card uses only the first 4K of video RAM, the CGA card uses the first 16K (one half of the available address range) and the Hercules card uses 32K and may extend into the second half of the B segment. The range beginning at B800:0000 acts as the second screen page. Changing DIP switches on the video card may be required, if range B800:0000 is being used by a color card. Also, EGA and VGA cards use a large portion of video RAM. Generally, these cards use up to 256K of video RAM; most of the first 32K of the B segment is used frequently. Section 4.8.2 demonstrates how to access the remaining portion of video RAM.

Addressing video RAM

Since video RAM is part of the PC's normal address space, usually it can be accessed just like normal RAM. The entire video RAM can be read with a screen refresh rate of 70 times per second by different components of the video card, which generates a video image.

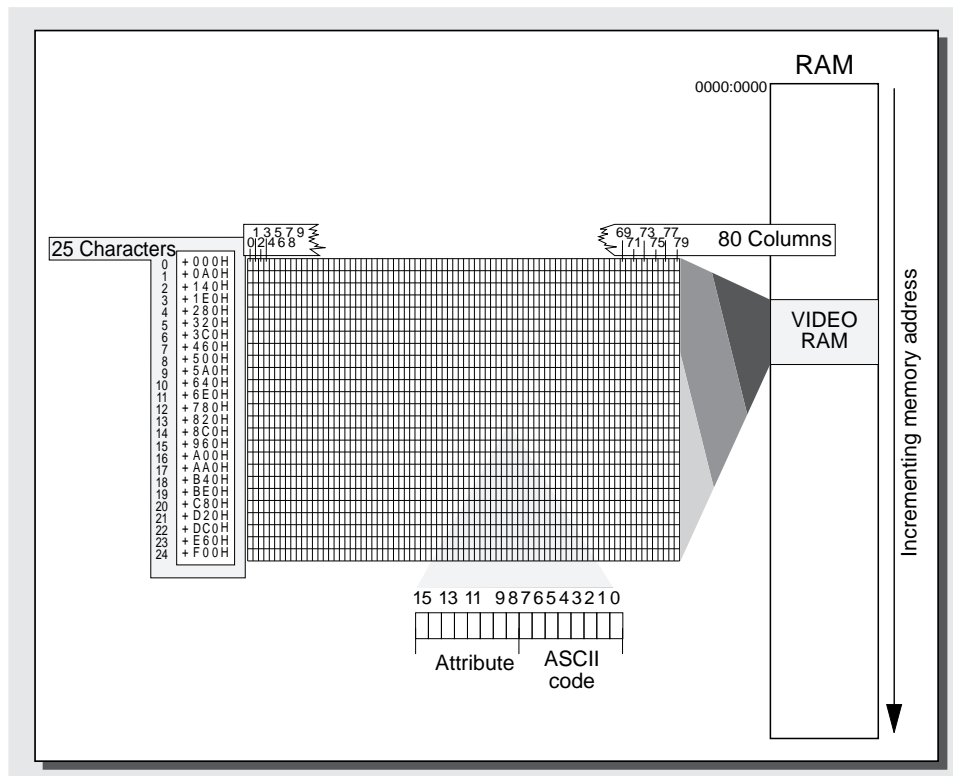
While this access isn't possible during an application or during ROM BIOS access, most video cards include options for avoiding memory conflict while accessing video RAM. The only exception is the older IBM CGA cards, which required some foresight in ROM BIOS or program access. (We'll discuss this in more detail in "The IBM Color/Graphics Adapter (CGA)" section later in this chapter.)

Video RAM structure in text mode

Most programs operate in text mode. However, some characters can be written directly to video RAM, regardless of the type of video card installed. Many programs, such as Lotus 1-2-3 and dBASE IV, use this option, with the help of equivalent BIOS functions. At the end of this section you'll find programs in Pascal and C. Both programs demonstrate direct access to video RAM in text mode. You could improve the speed even more by writing the code in assembly language. However, the routines are already quite fast. To understand how these routines work, you'll need a visual idea of how video RAM works, as demonstrated in the figure on the next page.

As you can see from the following illustration, in video RAM each screen position is represented as two bytes. The ASCII code of the character to be displayed is placed in the first of these two bytes (the one with the even address). By using eight bits per character code, a maximum of 256 characters can be displayed. See "The Video BIOS" section earlier in this chapter for more information on the PC character set.

*Normal video
RAM structure
in text mode*



The attribute byte (which defines the appearance of the character on the screen) follows the ASCII code. The attribute byte always appears at the odd offset address. The attribute controller subdivides this byte into two nibbles. The most significant nibble (bits 4 to 7) contains the character background and the least significant nibble (bits 0 to 3) describes the character foreground. These nibbles contain two values between 0 and 15, depending on the type of monitor involved. A color monitor (CGA or EGA) can display up to 16 possible colors on the screen. Each character has its own foreground and background.

Character organization in video RAM

To access video RAM, you must know how the individual characters are organized within this memory. This organization is similar to character display on the screen.

The first character on the screen (the character in the upper-left corner) is also the first character in video RAM, located at offset position 0000H. The next character to the right is located at offset position 0002H. All 80 characters of the first screen line follow in this manner. Since each screen character requires two bytes of memory, each line occupies 160 bytes of RAM. The first character of the second screen line follows the last character of the first line, etc.

Finding character locations in video RAM

You can easily find the starting address of a line within video RAM by multiplying the line number (starting with zero) by 160. To get from the beginning of the line to a character within the line, the distance of the character from the start of the line must be added to this value. Since each character requires two bytes, this is done by simply multiplying the column number (also starting at zero) by two. Add both products together to obtain the offset position of the character in the video RAM. These calculations can be combined into a single formula:

$$\text{Offset_position}(\text{row}, \text{column}) = \text{row} * 160 + \text{column} * 2$$

The RAM memory of the video card is integrated into the normal RAM of the PC system, so you can use normal memory access commands to access video RAM. You must know the segment address of video RAM, which is used together with the previous formula to find the offset position. Later we'll show you how this can be done easily in assembly language, Pascal and C.

NOTE

Since only 40 characters per line are displayed in 40-column video modes, you must use the value 80 instead of 160.

Now that we've discussed the most important similarities between video cards, in the following sections we'll describe the capabilities of these cards. Also, we'll explain how these capabilities can be used for optimal screen output.

Extended text modes

CGA, EGA and VGA cards recognize more than 80x25 text mode. CGA cards also have 40x25 text mode and EGA and VGA cards have 80 character text mode with more than 25 lines. The standard for EGA is 43 lines and the standard for many VGA cards is 50 lines. We can also include tricks for displaying 43 lines on VGA. If that isn't sufficient, Super VGA cards have various text modes, with resolutions of up to 132x80 characters.

However, these modes don't affect the video RAM structure. The more lines that we want displayed, the more calculations are involved in the offset address for adapting a character to video RAM. It should be sufficient to adapt a factor of 160, change the line width to characters and multiply that number by 2. However, 40x25 character text mode on a CGA card should use a factor of 80 instead of 160.

Occasionally, you won't have to program the video card. For example, a program run from DOS will usually start in 80x25 character text mode.

Access to different screen pages

As we mentioned, Hercules, CGA, EGA and VGA cards support multiple screen pages because video RAM offers much space for allocating a video page. To access multiple screen pages, use the formula discussed earlier and add the respective page to the offset address and starting offset.

We can organize the different screen pages in 40x25 text mode in 4K (4096 byte) units. Since a single screen page occupies 4000 bytes (80*25*2), 96 bytes are unused and ready for other applications. The offset formula for screen pages and 80x25 text mode is similar to the following:

$$\text{Offset_position}(\text{column}, \text{row}) = \text{row} * 160 + \text{column} * 2 + \text{page} * 4096$$

The first screen page contains a value of 0. Since the screen pages will be placed in 2K units, 40x25 text mode requires a factor of 2048 instead of 4096. If you're working with EGA and VGA text modes of 43 or 50 lines, then you must use a factor of 8K.

Sample programs

The following programs implement the above formula as a routine to transmit a string directly from video RAM to the screen. This is a small example, as many applications use multiple routines for direct video RAM access. These routines can include coloration of a screen range, storing screen contents in a window and restoring these contents later.

The three programs in BASIC, Pascal and C contain the same general components, but you'll notice there are some major differences in the way these programs are coded because of language differences.

You'll find the display routine itself, as well as an initialization routine for accessing the segment address of video RAM. The program determines whether a monochrome card or color card is installed in the system, by examining one of the BIOS variables (see Chapter 3). So at first glance, the video RAM segment address doesn't seem to have anything to do with it, because it contains the address of the CRTC address register. Upon closer examination, however, you will notice a direct connection between these two bits of information:

On monochrome cards this register is always located at port address 3B4H and the video RAM is always at segment address B000H. This connection between the port address of the address register and the segment address of the video RAM also applies to color cards, where the address register of the CRTC is always found at port address 3D4H and the video RAM is at segment address B800H. By determining the port address of the CRTC data register then, you find a statement about the segment address of the video RAM at the same time. Once you have determined this address, it is placed in a global variable and the initialization routine ends.

Along with this routine, all three programs have the actual output routine, which uses the segment address previously determined to access the video RAM. On top of that, the routine determines the start address of the screen page being displayed on the screen every time it is called. This is supposed to guarantee the output also appears on the screen and is not redirected to a screen page that doesn't get displayed on the screen. The routine also uses a variable from the BIOS variable area. The variable CRT_START is located at the address 0040:004E and registers the offset address of the displayed screen page relative to the first screen page at offset address 0000H.

After determining this address, the video RAM can finally be accessed. However, if this is done within the program, it is dependent on the particular language to a great extent. Let's take a look at the programs.

The BASIC implementation

The BASIC version of this program performs the required task. The execution of this program is slow, because of the alternative means of display. However, this program is a good example of how BASIC can take full advantage of the 80x86 memory system. DEF SEG, PEEK and POKE perform much of this memory access.

While DEF SEG always defines the segment address of the "current" 64K segment, PEEK and POKE lets you read the contents of a byte (PEEK) and write new contents to a byte (POKE). The InitDPrint routine uses this technique to define the

current segment as the BIOS variable segment. After defining the segment, two PEEK commands read the port address of the CRTC address register and changes the contents of the VSEG variable to the segment address of video RAM.

Once the current video RAM segment is defined, the offset address of video RAM must be defined as well. From this address the program takes the screen page from the BIOS variable range and adds this page to the offset of the display position within video RAM. The row coordinates (the ScRow% variable) are multiplied by 160 and the column coordinates (the Column% variable) are multiplied by 2.

The Pascal implementation

The ABSOLUTE statement and assembler routines allow Turbo Pascal to handle video RAM as a normal variable. Turbo Pascal provides a simpler method.

Turbo Pascal's MEM and MEMW functions allow Turbo to access memory ranges with known offset and segment addresses outside the Turbo Pascal data segment MEM handles bytes, while MEMW handles words. The two arrays are virtual (i.e., they don't really exist) and cover the entire memory range.

Values can be read from and written to these arrays, using the following syntax:

```
MEMW[ Segmentaddr. : Offsetaddr. ] := Expression
```

or

```
Variable := MEMW[ Segmentaddr. : Offsetaddr. ]
```

We use the MEM array in a display procedure which takes converted ASCII characters and a constant attribute. The DPrint procedure uses the MEMW array, as one 16-bit access on a 16-bit PC runs much more quickly than two consecutive 8-bit accesses on the same machine.

The MEMW array in DPrint takes the video RAM segment address from the VSeg variable initialized at the beginning of the program in the InitDPrint procedure. This procedure checks the contents of the BIOS variable contained in the port address of the CRTC address register. This is declared like other BIOS variables, called from within DPrint. Turbo's ABSOLUTE function prepares both variables, then handles them as global variables.

During DPrint's access to video RAM, the screen page number and coordinates are taken from the offset address of the MEMW array. From this information, the row coordinates are multiplied by 160 and the column coordinates are multiplied by 2. The string being processed is incremented by 2, shifting the paired ASCII attribute bytes to the right.

The Write statement and corresponding procedures write the text to the screen, provided the Crt unit is linked to the program and provided that DIRECTVIDEO is not set to FALSE.

The C implementation

This is the neatest of the three solutions, because the video RAM is treated as a normal variable. First, the VELB structure is defined, which describes the ASCII attribute pair as found in the video RAM. A new data type called VP is formed as pointers to this structure. It is important these be FAR type pointers, since these structures are within the video RAM, placing them outside of the C data segment. It is impossible to reach them in smaller memory models with their NEAR addressing without explicitly specifying the command word FAR.

You'll find the following program(s) on the companion CD-ROM



DVIB.BAS (BASIC listing)

You'll find the following program(s) on the companion CD-ROM



DVIP.PAS (Pascal listing)

The VPTR global variable within the INIT_DPRINT initialization routine is created as a pointer to the first ASCII attribute pair in page 0 of the video RAM. Within the actual output routine, it serves the DPRINT function as a basis for addressing the characters within the video RAM.

The LPTR pointer is loaded in the DPRINT output function with the address of the passed output position in the screen. First, the pointer is loaded with the contents of the VPTR global variable, with the offset address of the displayed screen page (from the variables in the BIOS region) being added to it.

Don't forget the LPTR pointer must first be cast in a BYTE pointer, since the contents of the BIOS variables don't refer to the VELB structure, but rather to Byte. Without the appropriate CAST operator, the C compiler would generate a code that multiplied the contents of the BIOS variables by the length of the VELB structure (2 bytes) before the addition, giving us the wrong result.

The actual output position, transmitted to the DPRINT function in the form of a Y- and an X-coordinate, must be added to the pointer. The video RAM is viewed as a kind of vector, whose 2000 components consist of the VELP structure. Since you have already determined the base address of this vector in LPTR, all that's left is to determine the index in this vector. Multiply the X-coordinate times 80 (columns per line) and then add the Y-coordinate. As an end result, you get a pointer to the output position in the video RAM, which can be treated like any other C pointer.

This pointer executes the specified string by pointing to the next VELB structure on each loop run. On each execution of the loop, the next ASCII attribute pair is placed in video RAM. The DPRINT function writes the ASCII code and character color to the specified string. This process repeats until the last character in the string is reached.

You'll find the following program(s) on the companion CD-ROM



DVIC.C (C listing)

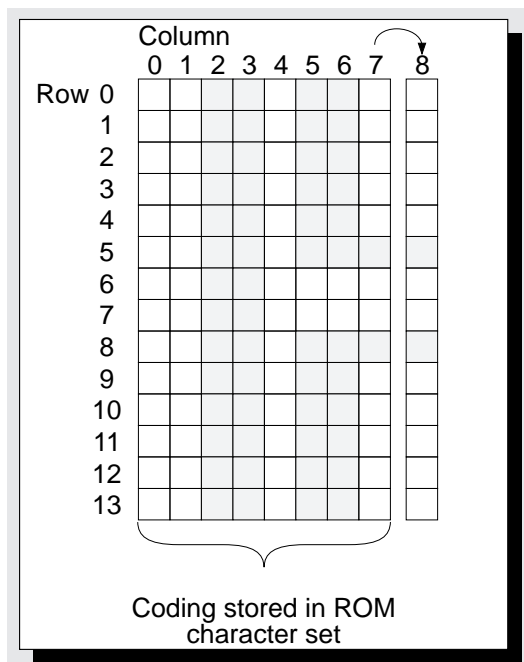
The IBM Monochrome Display Adapter (MDA)

The IBM Monochrome Display Adapter (or simply, MDA) is probably the oldest video card. This card is based on the Motorola 6845 video controller, which is an intelligent peripheral chip. The 6845 controller constructs a display by generating the proper signals for the monitor from video RAM.

This card is excellent for text display because of its 9x14 character matrix, which permits high resolution character display. The format of this matrix is unusual since a character generator containing the bit pattern of each character can only produce characters that are 8 pixels wide. Characters from the IBM character set may not connect with each other (e.g., using box characters to draw a box). A circuit on the graphics card avoids this problem by copying the eighth pixel of the line into the ninth pixel for any characters whose ASCII codes are between B0H and DFH. This enables the horizontal box drawing characters to connect.

The character generator requires one byte for each screen line: one bit per pixel, eight bits per line. Each character requires 14 bytes. The complete character set has a

*Monochrome Display Adapter
9x14 character matrix*

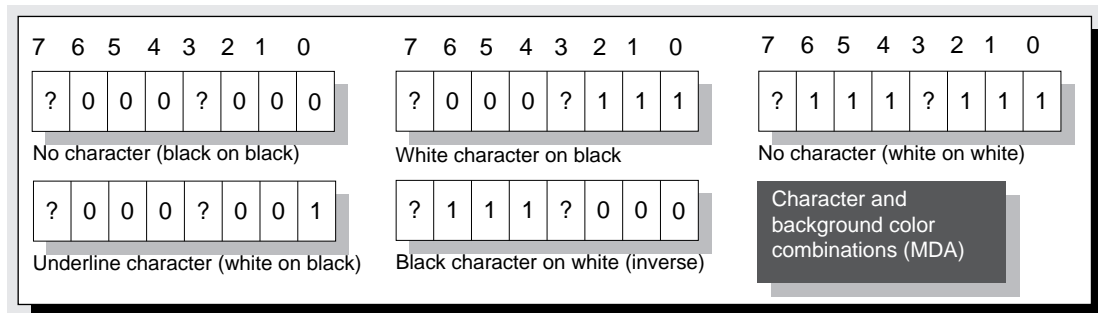
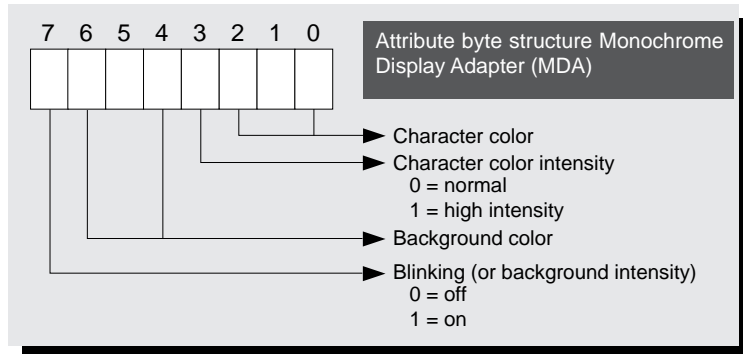


memory requirement of almost 4K, which is stored in a ROM chip on the card. For some reason the card has an 8K ROM, so the second bank of 4K remains unused.

Video RAM on the MDA

The video RAM of the card starts at address B000:0000 and extends over 4K (4,096 bytes). Since the screen display only has space for 2,000 characters and requires only 4,000 bytes of memory for those characters, the unused 96 bytes at the end of video RAM are available for other applications.

The illustration on the right shows the meanings of the different values representing the attribute byte. Any combination of bits can be loaded into this byte. However, the MDA only accepts the following combinations:

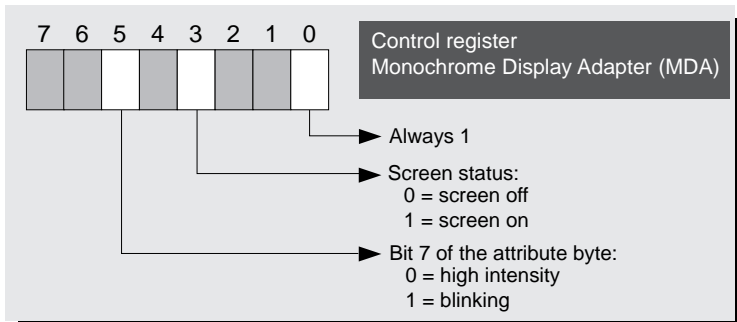


Besides these bit combinations, bits 3 and 7 of the attribute byte can be set or unset. Bit 3 defines the intensity of the foreground display. When this bit is set, the characters appear in higher intensity. Bit 7's purpose varies with the contents of the control registers (more on this later). For now, all you need to know is that bit 7 can either enable blinking characters or enable an intensity matching the background color.

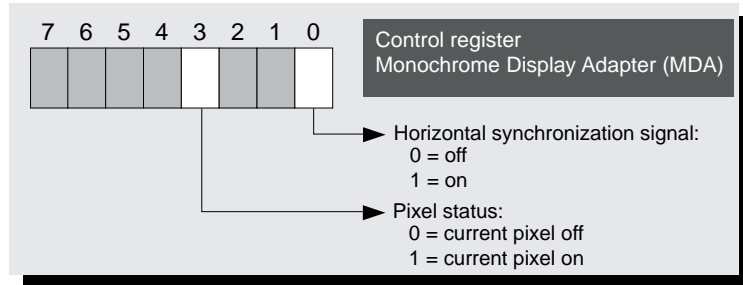
The control register and the status register are also available for monochrome cards.

MDA control register

The MDA control register, which is located at port 3B8H, controls the monochrome display adapter's various functions. As the following figure shows, only bits 0, 3 and 5 are important. Bit 0 controls the resolution on the card. Although the card only supports one resolution (80x25 characters), this bit must be set to 1 during system initialization. Otherwise, the computer goes into an infinite wait loop. Bit 3 controls the creation of a visible display on the monitor. If bit 3 is set to 0, the screen is black and the blinking cursor disappears. If bit 3 is set to 1, the display returns to the screen. Bit 5 has a similar function. If bit 7 in the attribute byte of the character is set to 1, it enables blinking characters. If bit 7 contains the value 0, the character



appears, unblinking, in front of a light background color. This means that bit 7 of the attribute byte acts as an intensity bit for the background. This register can only be written. This makes it impossible for a program to determine whether the display is switched on or off. The normal value for this register is 29H, which indicates that all three relevant bits default to 1.



MDA status register

Only bits 0 and 3 are used in the MDA status register; all the other bits must contain the value 1. Unlike the control register, programs can read this register, but register contents cannot be changed by program code.

Horizontal synchronization

Bit 0 indicates whether a horizontal synchronization signal is being sent to the screen. The video card sends this signal after creating a screen line (which is different than a text line, which consists of 14 screen lines) on the screen. This signal informs the electron gun, which "draws" the picture on the screen, that it should return to the left border of the current screen line. In this case, the bit has the value 1. Bit 3 contains the value of the pixel where the electron beam is currently located. A 1 signals the pixel is visible on the screen and 0 indicates the screen remains black at this location.

Accessing the CRT controller

The 6845's address register in the MDA card lies at port address 3B4H and the data register lies at port address 3B5H. Although these exist as consecutive port addresses, the number of registers to be addressed and the new contents of this register aren't output through port 3B4H by a 16-bit OUT instruction (this is the case with the other video cards). Also, the output must follow with the help of two 8-bit OUT instructions, between which a brief (5 or 6 cycles) pause is added to give the CRTC a chance to react to the output in the address register. Within an assembly language program, this pause executes and, after the OUT instruction, is followed by a jump instruction.

```
JMP $+2
```

CRTC registers in 80x25 text mode (Monochrome Display Adapter)					
Reg.	Meaning	Content	Reg.	Meaning	Content
00H	Total horizontal character	7	09H	Number of scan lines per screen line	13
01H	Display horizontal character	80	0AH	Starting line of blinking screen cursor	11
02H	Horizontal synchronization signal after ...char	82	0BH	Ending line of blinking screen cursor	12
03H	Duration of horiz. synchronization signal in char	15	0CH	Starting address of displayed screen page (high byte)	0
04H	Total vertical character	25	0DH	Starting address of displayed screen page (low byte)	0
05H	Adjust vertical character	6	0EH	Character addr. of blinking screen cursor (high byte)	0
06H	Display vertical character	25	0FH	Character addr. of blinking screen cursor (low byte)	0
07H	Vertical synchronization signal after ...char	25	10H	Light pen position (high byte)	*
08H	Interlace mode	2	11H	Light pen position (low byte)	*
*not available on MDA					

The transfer of program execution to the instructions that follow takes some time, but doesn't change the actual program execution. This time can be used by the CRTC to prepare access to the desired register.

Programming the CRTC register on this card consists of the starting and ending line of the blinking cursor and its position on the screen. These tasks can be easily accomplished using the function 10H sub-functions from the BIOS interrupts. This keeps hardware calls to a minimum. If you want to juggle CRTC registers and create different displays, such as an 81 column or 26 line screen, you can do this by manipulating CRTC registers in the MDA card while in 80x25 text mode.

The VMONO.ASM program which you'll find on the companion CD-ROM uses all the Monochrome Display Adapter's capabilities. It was written in assembly language.

The individual routines are completely documented and require no additional explanation. The demonstration program that's built into the listing demonstrates some practical ways to use the individual routines.

You'll find the following program(s) on the companion CD-ROM



VMONO.ASM (Assembler listing)

The Hercules Graphics Card (HGC)

In 1982, only a year after the IBM PC was released, Hercules Computer Technology made their contribution to the PC market by releasing the Hercules Graphics Card or HGC. Until then, only two video systems existed for the IBM: the MDA card for reading only text and the CGA for graphics display and fuzzy text. The Hercules card was intended for excellent text and graphics display. This card can drive a normal monochrome monitor in 80x25 text mode, with a 720x348 pixel graphics mode. The Hercules Graphics Card was an immediate success.

Non-BIOS support

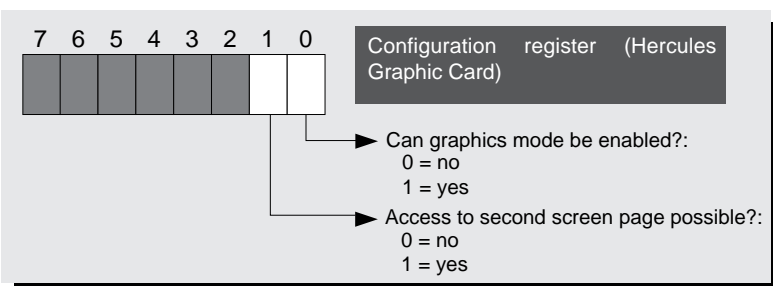
Since the IBM BIOS supports only the MDA and CGA cards, the Hercules card isn't supported by BIOS. This doesn't present a problem in text mode because the Hercules text mode is completely compatible with the MDA card. However, this nonsupport is noticeable in graphics mode because in this mode BIOS functions cannot set or read pixels. In this chapter you'll find some assembly language routines that solve these problems.

Video RAM on a Hercules Graphics Card

The Hercules card contains 64K of RAM, allowing it to operate in a memory intensive graphic mode. This RAM can be divided into two screen pages. Each of the two pages is 32K in length and can comprise either a text page requiring only 4K or a graphic page. The first screen page lies in the address range from B000:0000 to B000:7FFF. The second screen page follows, lying in the range from B000:8000 to B000:FFFF. Since this range matches those normally used by color video cards (CGA, EGA and VGA), this range can be adjusted by setting the DIP switches on the card. You can reconfigure the card to provide only one screen page.

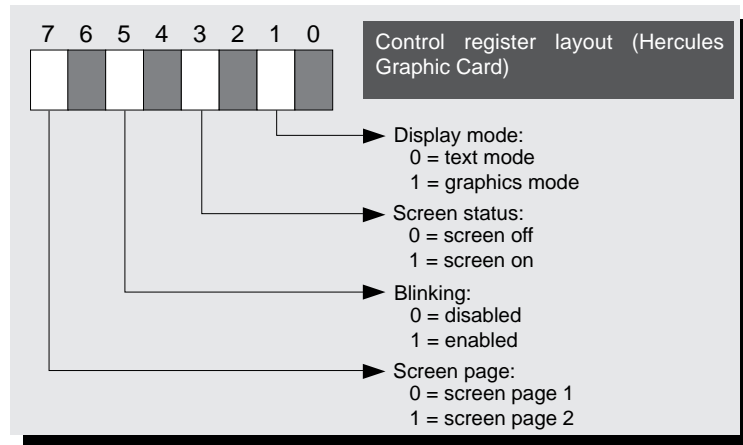
Unlike the MDA card, you can communicate with the configuration register through port 3BFH. You can write to this port, but you cannot read from it. This register has two bits (0 and 1). Bit 0 specifies whether graphic mode is enabled (1) or disabled (0). Bit 1 specifies whether the second screen page is available (0 if not, 1 if so).

To avoid conflicts with other video cards (color cards) both bits must be set to 0 so a graphic won't be displayed and the second screen page won't be used. If you want a program with these features to continue accessing the card, the control register must be changed accordingly.



The control register of the Hercules card differs from the control register of the MDA, which we've discussed.

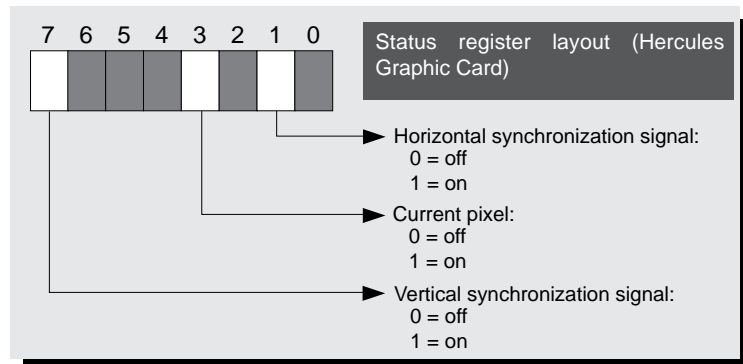
Unlike the IBM monochrome display adapter, bit 0 is unused and doesn't have to be set to 1 during the system boot. Bit 1 determines text or graphics mode: 0 enables text mode or 1 enables graphics mode. As you'll see in the following examples, changing these bits isn't enough for switching between text and graphics modes. The internal registers of the 6845 must also be reset. During this process, the screen display must be switched off to prevent the 6845 from creating garbage during its reprogramming.



The Hercules card has a seventh bit in this register. The contents of this bit determine which of the two screen pages appear on the screen. If this bit is 0, the first screen page appears and if it is 1, the second screen page appears. The user can write to or read from either page at any time. You can only write to this register; if you try to read this register, the value FFH is returned. Because of this, it's impossible to switch off the display simply by reading the contents of the status register and erasing bit 3, regardless of the display mode and the screen page selected.

Unfortunately, the control register cannot be read. This means that you won't know the card's current mode or when the card has initialized itself. This is a serious problem in TSR programs.

Unlike the MDA card, the meaning of bit 7 in the Hercules status register has changed. In the Hercules card this bit always contains a 0 when the 6845 sends a vertical synchronization signal to the screen, to generate a new screen structure.



The Hercules CRT controller

Similar to the MDA and CGA cards, the Hercules Graphics Card has a 6845 CRT controller as its main processor. Its address register is at port address 3B4H and its data register is at port address 3B5H. Unlike the MDA, the Hercules CRTC can be accessed by loading a 16-bit OUT instruction with the contents of the AX register. The output follows through the address register, where the processor register AL must contain the number of CTRC registers to be addressed and the AH register must contain the new contents of these registers.

The following table lists the values that must be loaded into individual registers to initialize text or graphics mode.

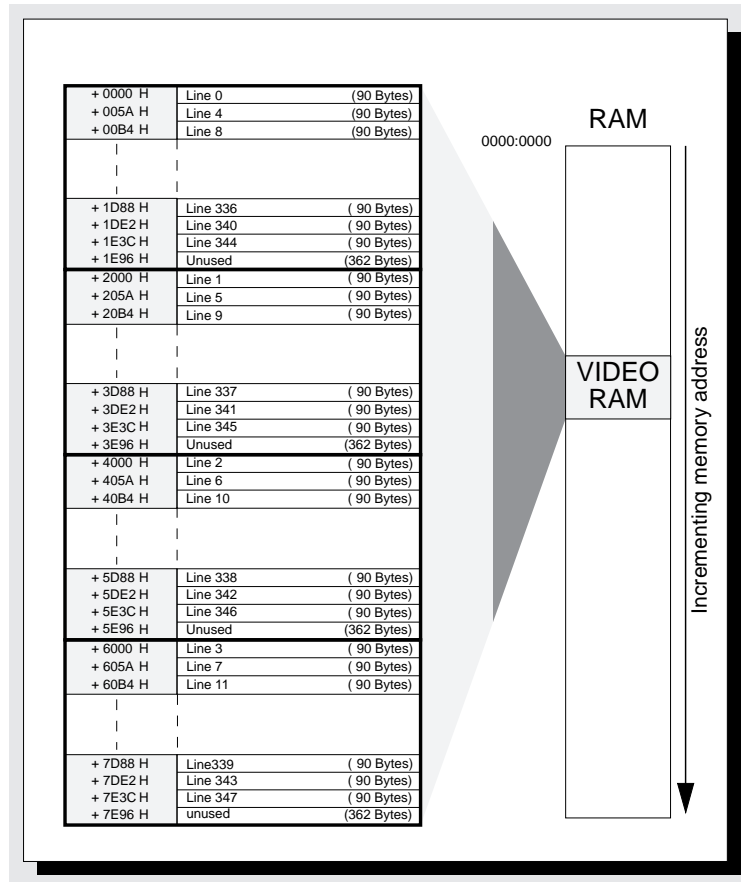
No.	Meaning	Text	Graphics	No.	Meaning	Text	Graphics
0H	Total horizontal character	97	53	9H	Number of scan lines per line	13	3
1H	Horizontal character displayed	80	45	AH	Starting line of blinking cursor	11	0
2H	Horiz. synchronization signal after character	82	46	BH	Ending line of the blinking cursors	12	0
3H	Horiz. synchronization signal width	15	7	CH	High byte of screen page starting address	0	0
4H	Total vertical character	25	91	DH	Low byte of screen page starting address	0	0
5H	Vertical character justified	6	2	EH	High byte of blinking cursor char. address	0	0
6H	Vertical character displayed	25	87	FH	Low byte of blinking cursor char. address	0	0
7H	Vert. synchronization signal after character	25	87	10H	Light pen position (high byte)	?	?
8H	Interlace mode	2	2	11H	Light pen position (low byte)	?	?

? = depends on light pen's position

Starting and programming graphics mode

You cannot switch to graphics mode using BIOS, although text mode automatically starts on bootup through the BIOS.

Arrangement of video RAM in graphics mode

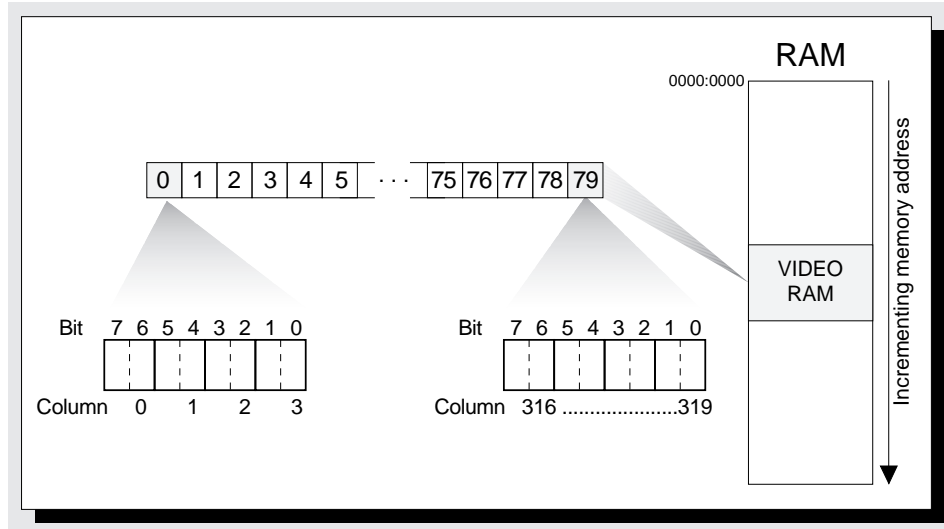


As we mentioned, the Hercules card in graphics mode provides 348x720 resolution. Each pixel on the screen corresponds to one bit in the video RAM. If the corresponding bit contains the value 1, the dot is visible on the screen; otherwise it remains hidden. The illustration above shows how video RAM is arranged in graphics mode.

The bit patterns of the individual lines in the video RAM aren't arranged sequentially. Instead, the 32K of video RAM is divided into four 8K blocks. The first block contains the bit pattern for any lines divisible by 4 (0, 4, 8, 12, etc.). The second block contains the bit patterns for lines 1, 5, 9, 13, etc. The third block contains the bit patterns for lines 2, 6, 10, 14, etc. and the last block contains lines 3, 7, 11, 15, etc.

When the 6845 generates a display, it obtains information for screen line zero from the first data block and screen line one from the second data block, etc. After it has obtained the contents of the third screen line from the fourth data block, it accesses the first data block again for the structure of the fourth line. Each line requires 90 bytes within the individual data blocks and each pixel requires a bit (720 pixels divided by 8 bits (per byte) equals 90). The first 90 bytes in the first memory area provide the bit pattern for screen line zero and the next 90 bytes provide the bit pattern for the fourth screen line. The zero byte of one of these 90-byte sets represents the first eight columns of a screen line (columns 0-8). The first byte represents columns 8-15, etc. Within one of these bytes, bit 7 corresponds to the left screen pixel and bit 0 corresponds to the right screen pixel.

*Relationship
between 90-byte
lines and screen
display*



Relationship between 90-byte lines and screen display

If the screen pixels of a line (0 to 719) and the screen pixels of a column (0 to 347) are sequentially numbered, an equation indicates the address of the bytes relative to the beginning of the screen page. This address contains the information for a pixel with the coordinates X/Y. To determine the bit within the byte that represents the pixel, the following formula can be used:

$$\text{Address} = 2000\text{H} * (\text{Y} \bmod 4) + 90 * \text{int}(\text{Y}/4) + \text{int}(\text{X}/8)$$

To send the number of desired bits within this byte, use the following formula:

$$\text{Bitnumber} = 7 - (\text{X} \bmod 8)$$

The VHERC.ASM program which you'll find on the companion CD-ROM demonstrates the capabilities of the Hercules Graphics Card. The individual routines within this program differ from the routines in the Monochrome Display Adapter demo program from the previous section. The routines here enable access to both screen pages and support the Hercules graphics mode.

You'll find the following program(s) on the companion CD-ROM



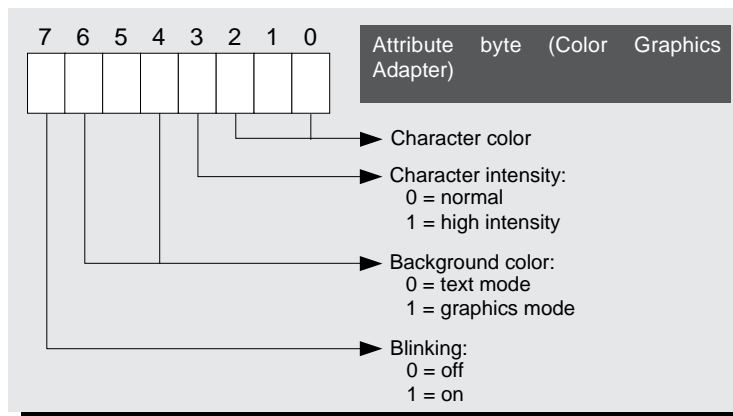
VHERC.ASM (Assembler listing)

The IBM Color/Graphics Adapter (CGA)

Similar to the MDA card, the CGA card was introduced in the early days of the PC. It was the standard for graphics output for many years until the EGA card replaced it. Unlike any other card, you cannot access the video RAM without the help of the CRT controller, without causing grainy pictures or snow.

CGA text modes

The CGA card recognizes two text modes, which comprise 40x25 characters and 80x25 characters. In reference to the video BIOS initialization function, these modes are assigned codes 1 (40x25) and 3 (80x25). These modes include two variants, which let you select the foreground and background colors from a palette of 16 colors. BIOS modes 0 (40x25) and 2 (80x25) send no color signal to the monitor, where the color codes are automatically converted to gray scales.



Each 80x25 text page requires 4,000 bytes of video RAM. 16K allows a total of four text pages. The first page starts at address B800:0000, the second at B800:1000, the third at B800:2000 and the last at B800:3000. The 40x25 mode allows storage of eight screen pages because, in this mode, each screen page only requires 2,000 bytes. The first screen page starts at address B800:0000, the second at B800:0800 and the third at B800:1000.

Attribute bytes

The lower four bits of the attribute byte indicate one of the 16 available colors. The

meanings of the upper four bits depend on whether blinking is active. If it's active, bits 4 to 6 indicate the background color (taken from one of the first eight colors of the color palette), while bit 7 determines whether the characters blink. If blinking is disabled, bits 4 to 7 indicate the background color (taken from one of the 16 available colors).

Color/Graphics Adapter color palette							
Dec	Hex.	Binary	Color	No.	Meaning	Text	Graphics
0	00H	0000(b)	Black	8	08H	1000(b)	Dark gray
1	01H	0001(b)	Blue	9	09H	1001(b)	Light blue
2	02H	0010(b)	Green	10	0AH	1010(b)	Light green
3	03H	0011(b)	Cyan	11	0BH	1011(b)	Light cyan
4	04H	0100(b)	Red	12	0CH	1100(b)	Light red
5	05H	0101(b)	Magenta	13	0DH	1101(b)	Light magenta
6	06H	0110(b)	Brown	14	0EH	1110(b)	Yellow
7	07H	0111(b)	Light gray	15	0FH	1111(b)	White

Graphics modes

The CGA supports three different graphics modes, of which only two are normally used. The color-suppressed mode; displays 160x100 pixels with 16 colors. The 6845 supports this resolution, but the rest of the hardware doesn't offer color-suppressed mode support. The remaining two graphic modes have resolutions of 320x200 and 640x200 respectively. The 320x200 resolution permits four-color graphics, while 640x200 resolution only allows two colors.

320x200 resolution

The CGA uses all 16K of its video RAM for displaying a graphic in 320x200 resolution with four colors. This limits the user to one screen page at a time. Of the four colors permitted, the background can be selected from the 16 available colors. The other three colors originate from one of the two user-selected color palettes, which contain three colors each.

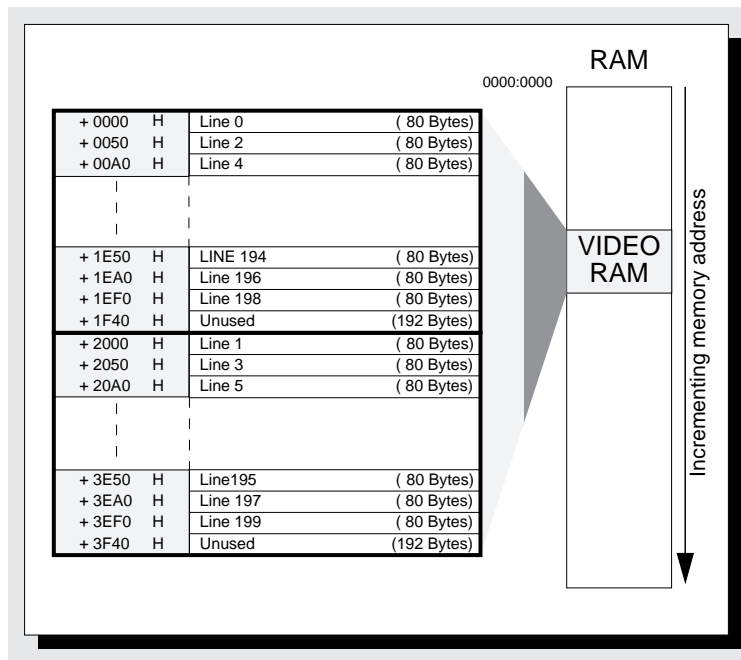
Palette 1:	Color 1	Cyan	Palette 2	Color 1	Red
	Color 2	Violet		Color 2	Green
	Color 3	White		Color 3	Yellow

Since a total of four colors are available, each screen pixel requires two bits. Four bits can represent the color numbers (0 to 3). The following values correspond to the various colors:

0	00(b) = freely selectable background color	1	01(b) = color 1 of the selected palette	2	10(b) = color 2 of the selected palette	3	11(b) = color 3 of the selected palette
---	--	---	---	---	---	---	---

The video RAM assignment in this mode is similar to that of the Hercules card during graphics display. The individual graphic lines are stored in two different blocks of memory. The first block, which begins at address B800:0000, contains the even lines (0, 2, 4,...); the second block, which begins at B800:2000, contains odd lines (1,3,5).

Arrangement of video RAM in graphics mode (blocking)

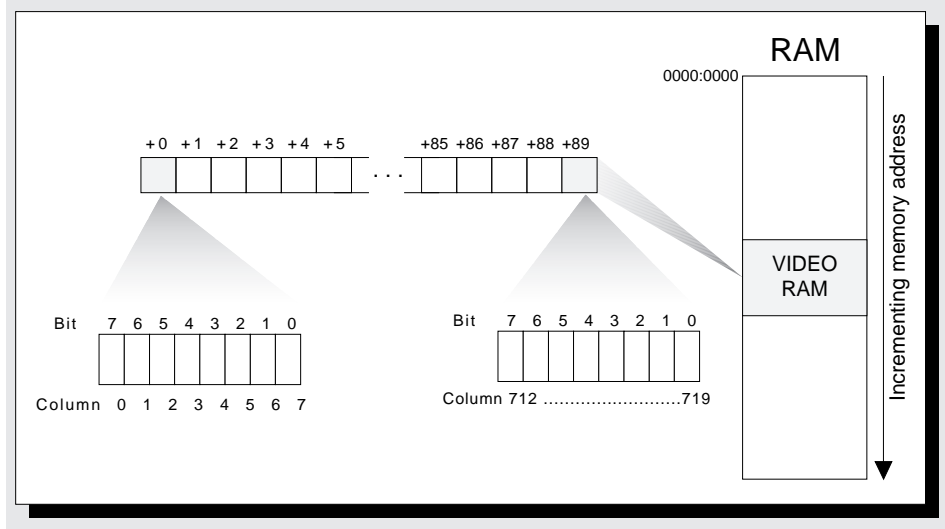


The desired palette can be accessed either by direct programming of the color selection registers or by calling a special BIOS function (interrupt 10H, function 0BH, sub-function 01H). Load the AH register with the function number (0BH), the BH register with the sub-function (01H) and the BL register with the number of the desired palette (0 or 1).

Each graphic line within the two blocks requires 80 bytes, since the 320 pixels in a line are coded into four pixels to a byte. The first byte in a graphic line (an 80-byte series) corresponds to the first four dots of the graphic on the screen. Bits 7 and 8 contain the color information for the leftmost pixel, while bits 0 and 1 contain the color information for the rightmost pixel of the byte. With this information, you can derive a formula for determining the byte in video RAM, similar to the Hercules card. This byte is relative to the starting address of the screen page, which contains the color information for a pixel. The screen column (0-319) is designated as X and the screen line (0-199) as Y:

$$\text{Address} = 2000H * (Y \bmod 2) + 80 * \text{int}(Y/2) + \text{int}(X/4)$$

*Representation
of a graphic line
in 320x200
resolution*

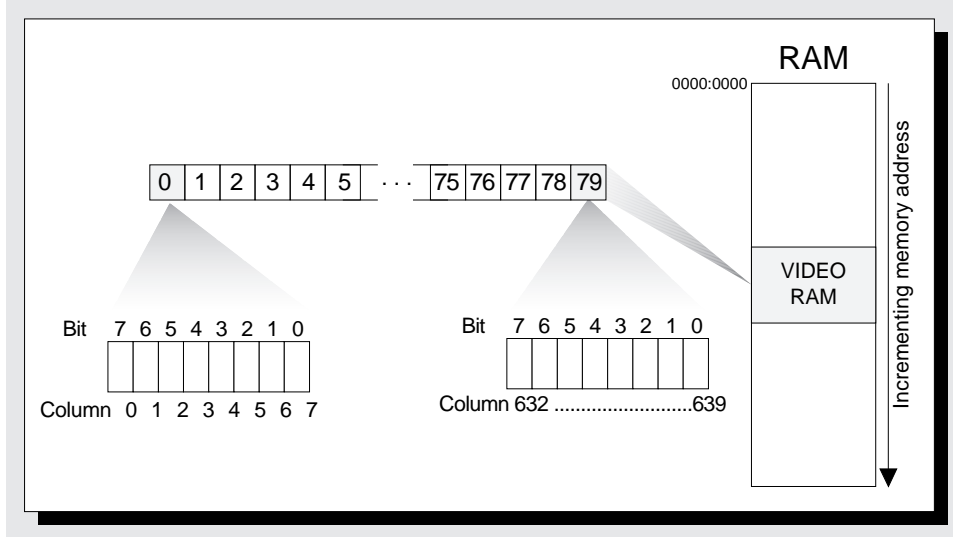


To determine the number of the two bits within this byte that represents the pixel, use the following formula:

$$\text{Bit number} = 6 - 2 * (X \bmod 4)$$

For example, if this formula returns 4, the color information for the dot is coded into bits 4 and 5. The 320x200 pixel graphics mode can also suppress the color signal, converting colors to gray scales using mode code 5.

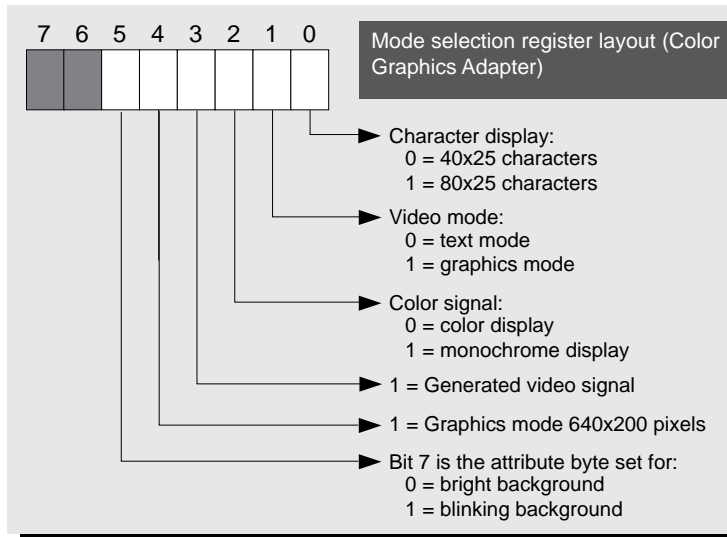
*Representation
of a graphic line
in 640x200
resolution*



640x200 resolution

The 640x200 pixel mode (BIOS code 6) is similar to the 320x200 pixel mode previously described. The doubled resolution makes only one bit per graphic pixel available, limiting the display to two colors. However, this mode has both odd and open

lines in two different memory blocks and the width of a graphic line in video RAM is 80 bytes. Addressing a pixel is identical to the 320x200 pixel mode.



$$\text{Address} = 2000\text{H} * (\text{Y} \bmod 2) + 80 * \text{int}(\text{Y}/2) + \text{int}(\text{X}/8)$$

$$\text{Bit_number} = 7 - (\text{X} \bmod 8)$$

CGA registers

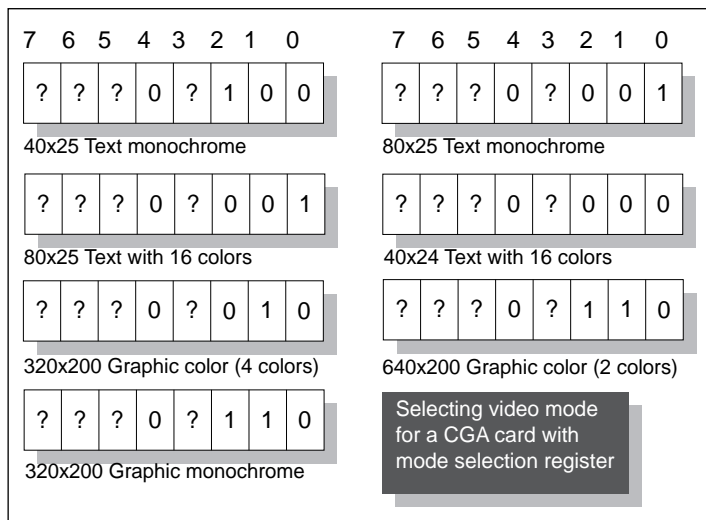
The CGA has a selection register at address 3D8H, which is comparable with the control register of the monochrome display adapter. You can write to this register but you cannot read it. Pixels that represent bits with values of 0 appear in this mode as black pixels. However, if a bit is set, the corresponding pixel appears on the screen as coded in one of the bottom four bits of the color selection register (more on this register later).

The desired color can be implemented without direct programming using a video BIOS function (interrupt 10H, function 0BH, sub-function 00H). Place the function

number (0BH) in the AH register, the sub-function (00H) in the BH register and the color (0 to 15) in the BL register.

Bit layout

Bit 0 of this register determines the text mode display of 80 or 40 columns per line. A 1 in bit 0 displays 80 columns, while a 0 in bit 0 displays 40 columns. The following illustration shows how these bits must be programmed to obtain certain modes.



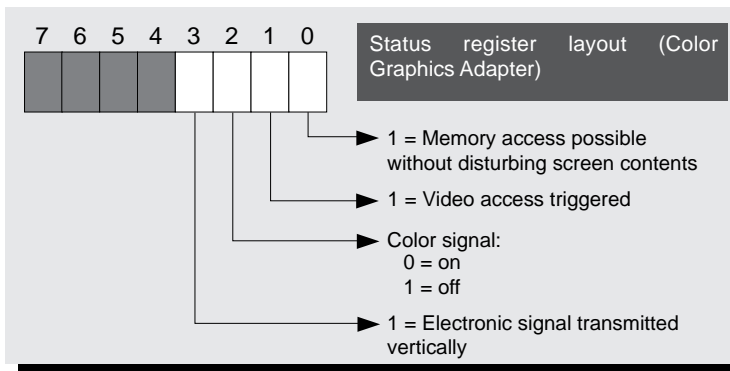
The status of bit 1 switches the CGA from text mode to the 320x200 bit-mapped graphics mode. A 1 in this register selects graphics mode, while a 0 selects text mode.

Bit 2 is useful if you want to use your CGA with a monochrome monitor. If this bit contains the value 1, the 6845 suppresses the color signal, displaying monochrome mode only. Bit 3 is responsible for creating screens. If this bit contains the value 0, the screen remains black. This suppression is useful when changing between display modes; it prevents sudden signals, which could cause damage, from reaching the monitor.

Bit 4 enables and disables 640x200 bitmapped graphics mode. A 1 in bit 4 enables this mode, while a 0 disables it. Bit 5 has the same significance as in the monochrome card. If it contains a 0, blinking stops and bit 7 returns

one of the 16 available background colors. This bit contains a default value of 1, which causes blinking characters.

The various text or graphics modes and the color or monochrome display can be selected in these modes with this register. Bits 0, 1, 2 and 4 are used for this for purpose.



The CGA also has a status register; similar to the status register in the monochrome display adapter. The illustration on the left shows the construction of this register, which can be found at address 3DAH. It's a read-only register.

Bit 0 of this register always contains the value 1 when the 6845 sends a horizontal synchronization signal to the monitor. This signal is transmitted when the creation of a line ends and the CRT's electron beam reaches the end of the screen line. The electron beam then jumps back to the left

corner of the screen line. This bit is significant because the CGA doesn't always allow data reading or writing within video RAM.

Flickering and the CGA

Flickering occurs because the 6845 must continuously access video RAM to read its contents for screen display. If a program tries to transmit data to video RAM, problems can arise when the 6845 accesses video RAM simultaneously. The result of this memory conflict is an occasional flickering on the screen.

To avoid this problem, you should only access video RAM when the 6845 isn't accessing it. This only occurs when a horizontal synchronization signal travels to the screen because it needs some time until the electron beam has executed this instruction. Therefore, the status register must be read before every video RAM access on a CGA. This process must be repeated until bit 0 contains the value 1. When this happens, a maximum of two bytes can then be transmitted to video RAM.

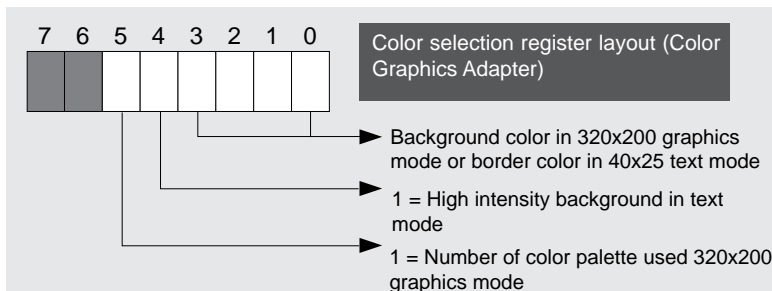
Sample program

The program at the end of this section demonstrates how this process works. This delay in video RAM access doesn't occur with monochrome cards because they are equipped with special hardware logic and fast RAM chips. This is also true of most of the newer model color cards. Before waiting for the horizontal synchronization signal, which results in a delay of the display output, the user should try direct access to video RAM to test the color card's reaction time.

If many accesses to video RAM occur within a short period of time (e.g., scrolling the screen), the electron beam doesn't respond fast enough. The screen should be switched off using bit 3 of the mode selection register. This prevents the 6845 from accessing video RAM, which allows unlimited user access to video RAM. When data transfer ends, the screen can be switched on again. BIOS uses this method during scrolling, which results in the flickering "strobe effect".

Color selection register

The color selection register is located at address 3D9H. This register is write-only (cannot be read). The meanings of the individual bits in this register depend on the display mode. Text mode uses the lowest four bits for assigning the background color from the 16 available colors. In



320x200 graphics mode, these four bits indicate the color of all pixels represented by the bit combination 00(b) (background color).

Synchronizing screen access

To minimize flickering, the CRT controller and video RAM must be synchronized. Bit 0 in the CGA card's

status register helps you do this. If bit 0 contains a value of 1, the electron beam is performing a horizontal scan. If pixels aren't currently being displayed on the screen, the CRT controller isn't accessing video RAM. So, it can place a program in video RAM undisturbed.

To use this mechanism, every CGA access to video RAM should start with a short code sequence, consisting of two loops and end when the CRTC performs a horizontal rescan. The first loop reads the contents of the status register as long as a horizontal rescan doesn't occur (i.e., bit 0 contains 0). If this condition occurs, the second loop, which reads the contents of the status register, executes. This continues until bit 0 contains 1 and the system performs a horizontal rescan.

At first this procedure may seem very complicated. Although it's not immediately clear why the first loop waits for a horizontal rescan, it's important to know what happens when the rescan occurs. This loop ensures the second loop ends at the beginning of a horizontal retrace. If the first loop failed, then the second loop would end before it was supposed to end. However, then a conflict with the CRTC may occur during video RAM access.

Linking this routine doesn't guarantee access to video RAM because when the horizontal retrace ends, the video RAM access ends about 4 microseconds later. How many bytes can be transferred between video RAM and the processor in that time depends on the processor speed and other factors, such as data bus width. Let's see how this works on a basic PC with a system timer of 4.77 MHz. Exactly two bytes (an ASCII character and its attribute) can be transferred during the horizontal retrace.

In addition to horizontal retrace, you can also use the 7 millisecond pause during vertical retrace to access video RAM. Up to 800 bytes can be transferred during this period. However, vertical retraces are much less frequent than horizontal retraces. So, the best method is to simply transfer data during horizontal retraces.

Both procedures hinder screen output. However, many CGA cards don't even need these procedures. When video RAM is available, it can be used by both the CRTC and the processor. Your program code should be constructed so the code sequence mentioned can be suppressed by a flag. Although the possibility of flickering caused by the card cannot be tested by the program itself, the user can be given the option of calling the program using a switch to control this flickering. You'll see how we did this in the demonstration program at the end of this section.

We can also minimize screen flickering by completely suppressing video RAM access. This method is successful only if multiple video RAM accesses can occur sequentially (e.g., when scrolling the screen). So, a program could suppress video RAM access until it called BIOS video interrupt functions 06H (scroll up) or 07H (scroll down). To realize this, bit 3 of the mode select register would have to be switched off before the access and switched on after the access. The result is a "strobe effect" that occurs during scrolling or other functions.

Bit 5 selects the color palette for 320x200 mode. If this bit contains the value 1, the first color palette (cyan, violet, white) is selected. A value of 0 selects the second color palette (green, yellow, red).

The CGA card's CRT controller

The CGA card uses the 6845 CRT controller from Motorola (see Section 4.4 for more information on this controller). This controller accesses the 18 internal registers exactly like the MDA card, using two consecutive 8-bit OUT instructions. A single 16-bit OUT instruction (used in Hercules cards) isn't permitted. Unlike the MDA, the CGA's registers are at port address 3D4H and 3D5H. The following table shows the contents of the CRT register in different display modes:

Number	Register Name	Txt1	Txt2	Grfx
00H	Horizontal character total	56	11	56
01H	Horizontal characters displayed	40	80	40
02H	Horizontal synchronization signal to characters	45	90	45
03H	Horizontal synchronization signal in characters	10	10	10
04H	Vertical character total	31	31	12
05H	Vertical characters justified	6	6	6
06H	Vertical characters displayed	25	25	10
07H	Vertical synchronization signal to characters	28	28	11
08H	Interlace mode	2	2	2
09H	Number of scan lines per line	7	7	1
0AH	Starting line of blinking cursor	6	6	6
0BH	Ending line of blinking cursor	7	7	7
0CH	Screen page starting address (high byte)	0	0	0
0DH	Screen page starting address (low byte)	0	0	0
0EH	Cursor character address (high byte)	0	0	0
0FH	Cursor character address (low byte)	0	0	0
10H	Light pen position (high byte)	?	?	?
11H	Light pen position (low byte)	?	?	?
? = depends on light pen's position				

These registers are useful because they define the position and appearance of the cursor on the screen. In the "History And Highlights" section we described how to program these registers. The CGA adds registers 0CH and 0DH. These registers indicate the start of the video page, that must be displayed on the screen, as offset of the beginning of the 16K RAM on the card (B800:0000), divided by 2. Register 0CH contains the most significant 8 bits of this offset, while register 0DH contains the least significant 8 bits. Usually both registers contain the value 0, displaying the first screen page (beginning at the address B800:0000) on the screen. To display the first screen page, which begins at location B800:1000 in the 80x25 text mode, the value 1000H divided by 2 (800H) must be entered in both registers.

You'll find the following program(s) on the companion CD-ROM



VCOL.ASM (Assembler listing)

The VCOL.ASM demonstration program which you'll find on the companion CD-ROM accesses the Color/Graphics Adapter. The only significant difference between this program and the previous programs is the video controller can synchronize video RAM access and screen construction. This is necessary on all video cards where direct access to video RAM causes a flickering on the screen. The WAIT constant, defined directly after the program header, switches synchronization on or off. Its contents decide during the assembly of the program, whether to assemble the program lines for

synchronization listed in the source listing. Since these lines would slow down the screen considerably, they should be included only if it's absolutely necessary.

EGA And VGA Cards

In this section we'll discuss the features of EGA and VGA cards that separate them from their predecessors. Later in this section we'll discuss sprites because EGA and VGA cards can be used in animation programming. The most important feature in text output is the ability to work with different fonts. We'll discuss both the 16 color and 256 color VGA graphics modes and some tricks you can use to double VGA resolution in 256 color mode.

Although IBM set the standards with its original EGA and VGA cards, the newer cards from third party manufacturers surpass the performance and capabilities of the originals. While all cards adhere to IBM standards, there still isn't a standard for the expanded modes with resolutions up to 1024x768 pixels. So, each manufacturer has its own idea of what works best. We'll discuss this in more detail later in this book, try to find some common features and suggest some ways you can use them.

The complexity of EGA/VGA cards makes the direct programming of different controllers and registers more complicated. Programming becomes even more involved when manufacturers don't adhere to the standards set by IBM. As you'll see in this section, you won't always have to rely on direct programming to fully use these cards. Instead, programming can access the special functions supported by the special EGA/VGA BIOS.

Some of the subsections describe how individual registers are programmed for performing certain tasks. The last section summarizes all standard functions of the EGA and VGA registers. This section also includes a listing of expanded EGA and VGA BIOS functions. Before programming a register, check references to ensure the appropriate EGA/VGA BIOS function actually exists.

Monitors and cards

The kind of monitor used can significantly affect the performance of an EGA or VGA card. If your video card's capabilities exceed the physical capabilities of your monitor, you won't get the 800x600 resolution and billions of colors that may have been advertised. You'll need a more sophisticated monitor to use all the capabilities of your video card.

This applies to all the applications using your EGA or VGA card. Matching the capabilities of the video card and monitor is extremely important.

First, you must determine whether the monitor is color or monochrome. There are several possibilities with EGA and VGA cards.

EGA monitors

An EGA card may be connected to a CGA, EGA, multisync or monochrome MDA monitor. Depending on the type of monitor connected, the card behaves like either a normal EGA card or like an expanded MDA card. Switching between these modes affects the internal registers of the EGA card and video RAM. In monochrome mode, video RAM begins at B000H instead of B800H. Also, the attribute bytes of the characters in video RAM are interpreted in monochrome mode as they are with an MDA card. The CRT controller's index and data registers change to conform to the MDA standard.

Since an EGA card connected to a monochrome monitor behaves like an MDA card, we won't discuss this configuration in detail. When we mention an EGA card, we're referring to an EGA card with a high resolution monitor attached. This is usually the case because many EGA compatible cards don't even support monochrome monitors, even though the original IBM EGA card offered such support. Compatible cards work only with EGA or multisync monitors.

Another accessory that has almost vanished is the IBM EGA card with 64K of video RAM. These cards could be expanded to 128K, 192K and 256K. All newer EGA cards include 256K of RAM as standard equipment and we assume this standard configuration in the following sections.

VGA monitors

VGA cards operate differently. A VGA card can be connected to a monochrome monitor, but it must be an analog monochrome VGA monitor instead of a simple MDA monitor. Although analog VGA monitors cost less than color VGA monitors, they can still use the high resolution VGA modes. An analog VGA monitor isn't able to produce the entire VGA palette of 256 colors because it can display only 64 shades of gray. Usually this won't make a difference as long as you avoid colors that differ slightly in green components. Analog VGA monochrome monitors ignore the red and blue components of the RGB signal and use only the green component.

One advantage the monochrome VGA monitor has over its color counterpart is the VGA card can be switched into a monochrome mode, in which it behaves like a very powerful MDA card. Just as with the EGA card, the video RAM begins at B000H instead of B800H and the addresses of the various registers conform to the MDA standard.

Code	EGA/VGA video mode	MONO	CGA	EGA/VGA
00H	40x25 character text, 16 colors			
01H	40x25 character text, 16 colors			
02H	80x25 character text, 16 colors			
03H	80x25 character text, 16 colors			
04H	320x200 pixel graphics, 4 colors			
05H	320x200 pixel graphics, 4 colors			
06H	640x200 pixel graphics, 2 colors			
07H	80x25 characters, monochrome			
0DH	320x200 pixel graphics, 16 colors			
0EH	640x200 pixel graphics, 16 colors			
0FH	640x350 pixel graphics, monochrome			
10H	640x350 pixel graphics, 16 colors+			
11H	640x480 pixel graphics, 2 colors			*
12H	640x480 pixel graphics, 16 colors **			*
13H	320x200 pixel graphics, 256 colors			*
* only possible with VGA card ** EGA Cards with 64K of RAM can only display 4 colors				

You can switch modes on the card using the MODE MONO and MODE CO80 DOS commands or by using function 00H of the BIOS video interrupt. Only video mode 07H (which is also used by the MDA card) gives you monochrome operation. All other modes enable color operation on a VGA card. The large table on the left summarizes which monitors can display which EGA and VGA video modes.

Throughout this section we assume that your VGA card has already been switched to color mode for the demonstration program. For EGA cards, we'll always assume 256K of RAM and an EGA or a multisync monitor are being used.

Identifying EGA and VGA cards

If your programs use features that are unique to EGA or VGA cards and the performance depends on a certain monitor type, then your programs should include a query routine that checks for the required hardware before proceeding.

Our routine, called IsEgaVga, is listed as a function in both the C and Pascal versions of the demonstration program. The routine calls two BIOS functions that are found only in EGA and VGA cards. One of these functions immediately determines whether you have an EGA/VGA card or an older card (from MDA to CGA). The function fails if the card is a CGA or an MDA (i.e., the function doesn't exist on these cards).

The other function is available only on the VGA card, which then lets you distinguish between EGA and VGA. This is function 1AH, which has two sub-functions. We're interested in sub-function 00H, which returns information on active and passive video cards.

Active and passive video cards refer to a PC that may have a second video card, such as an MDA or a Hercules card, installed in addition to a VGA card. However, this doesn't apply to most PS2 models because MDA or Hercules cards aren't available for the MCA bus on these systems.

When you call the 1AH function with the function number in the AH register and the sub-function number in the AL register, you can tell by the contents of the AL register if the function is supported by the BIOS and if a VGA card is installed. A normal BIOS will simply leave the value 00H unchanged in the AL register, indicating the requested function isn't supported. The VGA BIOS loads the function number 1AH in the AL register to acknowledge the function call. The code returned to the BL register indicates the active video card and the code returned to the BH register indicates the passive video card. These codes are described in the following table:

Code	Meaning	Code	Meaning
00H	No video card	07H	VGA card with analog monochrome monitor
01H	MDA card with MDA monitor	08H	VGA card with analog color monitor
02H	CGA card with CGA monitor	09H	Reserved
03H	Reserved	0AH	MCGA with CGA card
04H	EGA card with EGA color monitor	0BH	MCGA with analog monochrome monitor
05H	EGA card with MDA monitor	0CH	MCGA with analog color monitor
06H	Reserved		

Code	EGA Video RAM	Code	EGA Video RAM
00H	64K	02H	192K
01H	128K	03H	256K

If the call to function 1AH fails, then we assume that a VGA card doesn't exist. In the next step, we check for an EGA card. Sub-function 10H of function 12H is supported only by EGA cards. Unlike many other BIOS functions, the sub-function number is passed

in the BL register instead of the AL register. If the sub-function number 10H remains in this register after the function call, the system contains an EGA card.

You'll find the following program(s) on the companion CD-ROM



ISEVP.PAS (Pascal listing)
ISEVC.C (C listing)

The contents of the BH register indicate if the EGA card is connected to an MDA or EGA color monitor. The value of this register is 1 for MDA and 0 for EGA. Also, the contents of the BL register indicates how much RAM is available, as indicated by the table above left.

The demonstration programs ISEVP.PAS and ISEVC.C each contain an IsEgaVga function and demonstrate its use within a program. You'll also find other slightly modified versions of this function in other demonstration programs later in this section. These programs

and program listings are on the companion CD-ROM.

Selecting and programming fonts

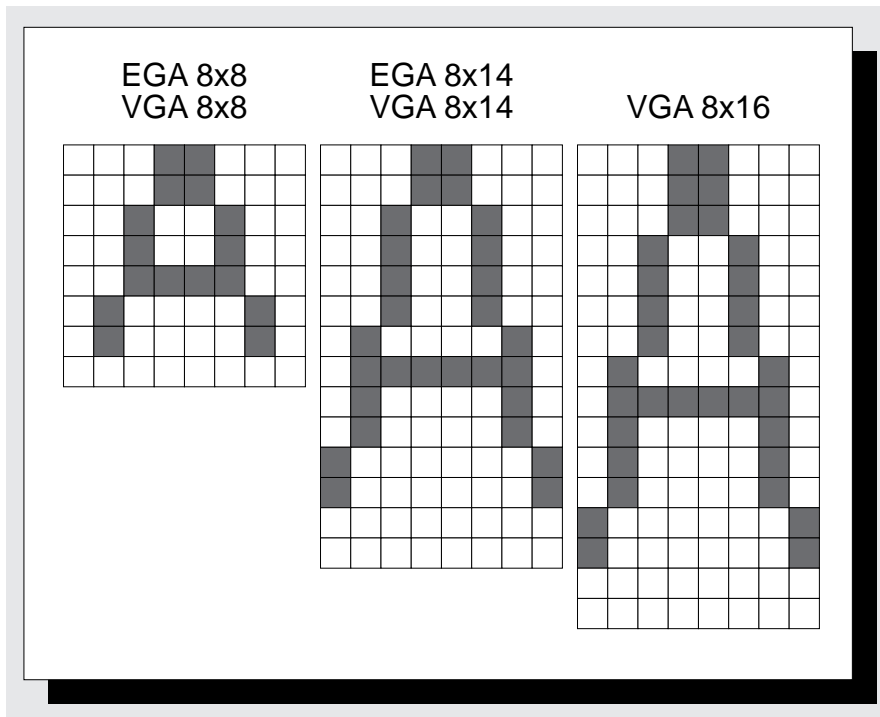
Unlike earlier video cards, EGA and VGA cards aren't limited to the fonts present in the ROM chips on the card. Instead, they use a powerful character generator that relates the characters in an area of the video RAM to the corresponding bitmaps used to display the characters. In this section we'll discuss the structure of the character table and explain how to use it.

Loading and defining fonts with the BIOS

The expanded EGA and VGA BIOS has several functions that allow you to manipulate character tables. There are more than a dozen sub-functions available through function 11H of the BIOS video interrupt. As usual, you must place the function number in the AH register and the sub-function number in the AL register when making the call.

In addition to working with predefined character tables, these functions also allow you to load your own character tables, display a character table on screen or switch between different character tables. The predefined character tables are stored in a ROM chip on the EGA or VGA card and copied from there to a portion of the video RAM. The character generator then accesses the information about the appearance of certain characters. EGA cards have only two fonts in ROM and VGA cards have three fonts. The third font on the VGA card can be activated through the BIOS.

*Character
structure within
a matrix*



The structure of the video fonts is different than the fonts that are used on a printer. The difference lies in the size of the characters (i.e., the matrix; size on which the characters are based). The EGA card normally uses an 8x14 pixel matrix for the first ROM font. The second ROM font uses a smaller matrix that squeezes the characters into an 8x8 pixel box. The pixels remain the same size, although the font changes in size. Since this means the characters in the 8x8 matrix are smaller and harder to read, you may be wondering why you would select a smaller font.

A smaller font occupies less space on the screen, which enables you to display more characters. The 8x8 font lets you display 43 lines of text instead of the usual 25, allowing you to display more information on a single screen.

EGA vertical resolution:	350 pixels	350 pixels
Character matrix height:	14 pixels	8 pixels
Number of text lines:	25 lines	43.75 lines

The number of text lines actually depends on the vertical resolution of the EGA card, which is always 350 pixels in text mode. From the calculations listed below, we see there are actually 43.75 lines available on the screen. But since displaying 3/4 of a line isn't possible, the number is rounded down to 43.

The horizontal resolutions for the two character matrices are the same. The EGA card always contains 640 pixels per line. Since each character matrix is 8 pixels wide, there are 80 characters per line.

The VGA card is slightly different in both horizontal and vertical resolution. The vertical resolution in text mode is 400 pixels instead of 350 pixels. So, the normal text screen of 80x25 characters is obtained with a special VGA character matrix. This character matrix has a height of 16 pixels. The two EGA fonts are also available on the VGA card and these fonts permit text resolutions of 28 and 50 lines.

The horizontal resolution of the VGA card is also higher with 720 pixels per line instead of 640 pixels per line. However, we still get only 80 characters per line in VGA text modes because the horizontal dimensions of the character matrices increase from 8 to 9 pixels. The ninth pixel has a special purpose, which we'll discuss later.

First, let's discuss the functions used to load a font. These functions also automatically specify the number of text lines that will be displayed on screen. The numbers of these functions are 11H, 12H and 14H. To call these functions, place the function number in the AX register and an additional argument in the BL register. This gives the number of the character table into which the selected font will be loaded and activated.

Sub-function	Matrix	EGA lines	VGA lines
11H	8x14	25	28
12H	8x8	43	50
14H	8x16		25

If you don't want to work with several character tables simultaneously, enter a value of 0 for the first character table. EGA cards can use values from 0 to 3 and VGA cards can use values from 0 to 7. The following sections provide additional information on the different character tables.

Sub-function	Matrix	EGA	VGA
1H	8x14		
2H	8x8		
4H	8x16		

These functions always load a certain font and then set the registers of the video card to display the corresponding number of text lines. It's also possible to simply load a font into a character table. If you want to work with a character table other than the one currently active, this won't affect the screen display because the font isn't activated and the number of text lines doesn't change. Calling this function

is identical to the three functions we just described except for the contents of the registers.

You aren't limited to the character tables stored in ROM; the BIOS also lets you load your own fonts. You can leave certain characters undefined or select different fonts, just as you can with a printer.

One feature allows you to set the character height from 1 to 32 pixels. The number of lines of text displayed on the screen is set accordingly. However, remember that characters less than 6 pixels high can't be read on the screen. Characters greater than 16 pixels high are easy to read, but this limits the number of lines you can display on screen simultaneously. Generally character heights should be from 8 to 16 pixels; otherwise an unusual looking screen display may be produced.

The BIOS lets you specify any number as the character height. Place the character height in the BH register when calling function 10H. The character width cannot be altered in any way; it remains fixed at 8 pixels for EGA mode and 9 for VGA mode.

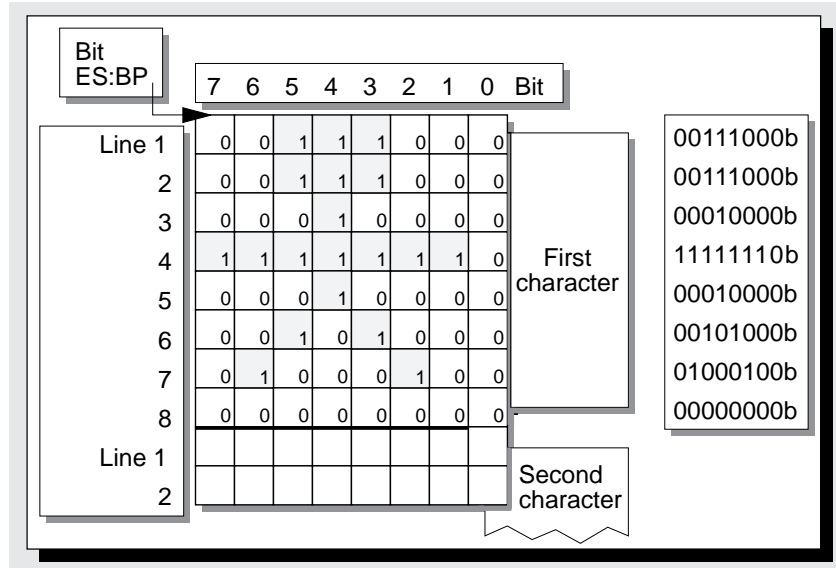
You can also select any character table for your font. The character table number is loaded in the BL register. The CX register is passed the number of characters to be loaded. This allows you to load only selected characters, if desired. In the DX register, you're expected to enter the ASCII code of the first character you want to load. This may be a number from 0 to 255.

The BIOS retrieves the pixel pattern data for the characters from a buffer. This buffer must be created and initialized by the function caller. The address of the buffer is expected as a FAR pointer in the ES:BP register pair.

As the previous figure shows, the buffer must have an entry for each character to be defined. The size of the entry in bytes corresponds to the character height. For example, if an 8x12 matrix has been defined, each buffer entry consists of 12 bytes. The total buffer size is then 12 times the number of characters to be defined, since the entries follow each other. The first entry has the offset address of 0000H. The data for the first character is found here, followed by the data for subsequent characters.

Within each entry, each byte represents the pixel pattern for one pixel line of the character. The bytes correspond to lines in ascending order from the first to the last line of the character. Within each byte, the individual bits indicate the status of each pixel in the line, from left to right. If the bit is set, then the corresponding pixel is displayed using the text foreground color. If the bit's off, then the pixel is displayed in the background color.

*Buffer structure
expected by
function 10H*



Sub-function 00H works the same as 10H. This function loads the given character definition in a character table, but doesn't activate the font or change the number of text lines on the screen. Function 00H corresponds to function 10H just as we saw functions 01H, 02H and 04H correspond to 11H, 12H and 14H.

When using characters and fonts that you've designed, don't forget they can be displayed only on your screen. Your printer won't be able to reproduce them. So, don't be surprised if your printer uses a completely different font if you try to print a screen displaying a custom font.

Changing from 9 pixel to 8 pixel display

The programs we'll discuss on the next few pages are examples of user-defined fonts applied to company logos and compound characters. However, this program has a problem with VGA cards because of the width of the characters. With an EGA card, each character has an actual width of 8 pixels. VGA characters are 9 pixels wide. This produces horizontal screen resolutions of 640 (EGA) and 720 (VGA).

You're probably wondering why there is a ninth pixel when the standard ROM fonts, as well as any font you can load with the BIOS, are 8 pixels wide. The ninth pixel is usually left blank. The only exceptions are characters with ASCII codes between C0H and DFH. This range contains various characters used for drawing frames and borders. These characters must be connected, without any space between them, to create unbroken horizontal lines. So, these characters simply copy the eighth pixel to the ninth to fill the empty space.

By adding the ninth pixel, VGA achieves higher resolution even though only 8 of the pixels are coded. This is possible because EGA actually uses only seven of the eight pixels. The eighth remains empty to ensure spacing between characters. VGA can use eight pixels and leave the ninth free.

As long as you define individual characters that have spaces between them, this characteristic of the VGA card doesn't usually cause any problems. But if you want to create a logo or a compound character (with two or more characters joined together), you'll encounter problems because you cannot control the ninth pixel. To work with such a font, you must sacrifice the additional resolution of the 9 pixel mode and use one of the normal 8 pixel modes as they are used with the EGA card.

This mode change involves several registers of the VGA card. Unfortunately, not all these can be addressed through the BIOS. Some must be programmed directly. First we'll discuss the miscellaneous output register. This register can be read using port address 3CCH and written using port address 3C2H. It contains two bits (bits 2 and 3) that set the VGA clock, which in turn determines the horizontal resolution.

In normal text mode, the VGA card runs at 28.322 MHz with 720 pixels per line. The frequency with 640 pixels per line is 25.175. This is the resolution we want to create a screen with 80 characters per line using 8 pixel characters. To switch your VGA card to the desired 640 pixel mode, we must modify bits 2 and 3 and then return the new register values. However, since the characters in VGA fonts are nine pixels wide, we must also switch to a character width of 8 pixels. This is done by changing the contents of the clocking mode register. This register is part of the sequencer controller and cannot be directly addressed like the miscellaneous output register. Before and after the sequencer controller register is accessed, it is reset using its reset register. This is done before a sequencer register, such as the clocking register, can be changed. Bit 0 of the clocking register is responsible for the character width.

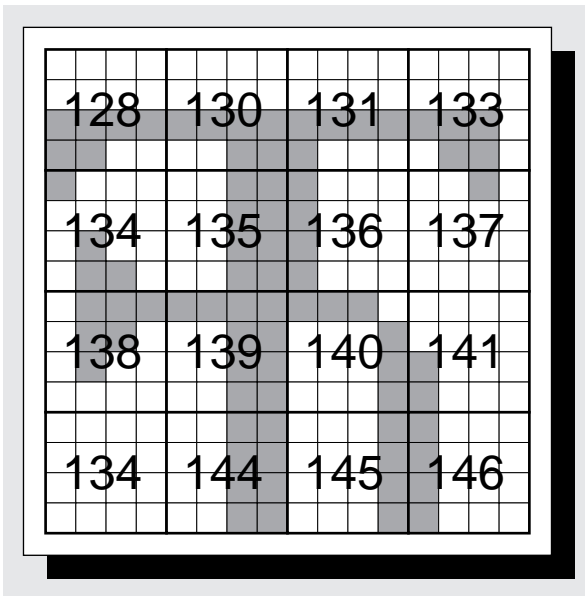
The last step is to modify the horizontal pel panning register. We've already encountered this register in a description of smooth scrolling. After switching to the 8 pixel width, this register is loaded with the value 0. This moves the picture one pixel to the left. If you don't do this, the first pixel column on the screen would flicker. We don't have to access this register directly, since there is a BIOS function available.

The three steps required to change from a 9 pixel character width to an 8 pixel character can also be reversed to revert to 9 pixel width. Simply switch the horizontal resolution back to 720 in the miscellaneous output register and set the character width back to 9 pixels in the clocking mode register. Don't forget to set the horizontal pel panning register back to 8 so the first pixel column is visible again. If you want to know the exact structure of these registers, refer to the end of this chapter. You'll find detailed descriptions of all EGA and VGA registers. The LOGOP.PAS demonstration program in the following section contains a routine for executing these changes.

Logos

Now let's discuss creating logos. Logos are an attractive way to introduce your software on screen. They give your software a professional look. For example, a logo, instead of a simple copyright message, can be displayed on the screen. However, it's difficult to create an interesting logo using only the standard characters available in the normal ASCII set. But both the EGA and VGA cards allow you to change the appearance of individual characters. Remember, if you have a VGA card, you must reduce the character width to 8 pixels.

*Logos from
compound
characters*



Usually, you need more than one character to create an interesting logo. Each character consists of 128 pixels on a VGA card (8x16) and only 112 on an EGA card (8x14). We'll explain how you can create logos by combining several characters into complex patterns.

Combining characters to create logos provides more pixels to work with and allows you to create a larger logo, which in turn demands more detail. However, you shouldn't use too many characters in your logo. Otherwise, you may lose too many useful characters from the ASCII character set. Also, you should ensure that all the characters continue to appear correctly so subsequent text is readable. Remember, to create the logo, we're redefining only a few characters instead of the entire font.

You should select characters that aren't used elsewhere in your program. Characters with ASCII codes less than 32, the foreign language characters and characters with ASCII codes greater than 224 are good choices. Depending on which characters your program needs, up to 100 characters are usually available for creating a logo. This allows you to create a 10x10 character logo with a resolution of more than 12,000 pixels.

We've included an ASCII table in the Appendix (located on the companion CD-ROM) which displays the possible choices. You'll see examples of the ASCII codes, including those described above, in this ASCII table.

To help you create your own logos, we've included the LOGO program in both C (LOGOC.C) and Pascal (LOGOP.P) versions. The main feature of each program is a function called BuildLogo function, which is responsible for the structure of the logo and for displaying it on screen. Data about the desired position of the logo on screen, the logo's height in pixels, its color and the string array that defines how the logo looks is required by this function.

In this array, each string represents one pixel line of the logo. Each character in the string represents a pixel in the line. If the character for a certain pixel is a space, then this pixel is blank. If any other character is used, the pixel is set. Although this method of storing the information for creating the logo uses a lot of memory, it also allows you to edit the appearance of the logo in the source code easily, without using a special editor. Also, since the logo's width is taken directly from the length of the string array, it doesn't have to be passed as an extra parameter.

Depending on the size of the logo and the video card that's installed (EGA or VGA), BuildLogo calculates the number of characters required and defines the pixel patterns for each individual logo character using sub-functions 00H and 10H of the BIOS video interrupt (previously described). If the logo doesn't completely fill the rectangle reserved for it, it is centered within it.

BuildLogo uses the foreign language characters to create logos. These are the characters with ASCII codes from 128 to 167; this provides a total of 39 characters. You can easily modify the BuildLogo function so you can also use other ASCII characters.

This function also switches the VGA character width from 9 pixels to 8 pixels. The IsEgaVga function determines whether the current system has a VGA card. The SetCharWidth function makes the switch. The steps required to switch the character width were already described.

BuildLogo defaults to character heights of 14 pixels on EGA cards and 16 pixels on VGA cards. The normal fonts are used, which produces a screen resolution of 25 lines by 80 columns. If you want to use BuildLogo in a program that uses a smaller font (resulting in better screen resolution), then you must change the value of the CharHeight variable in the LOGO programs. If you use a smaller font, remember that you'll need more characters to create the same logo because each character will have fewer pixels in its height.

If you no longer need a logo, the ..ResetLogo function; can restore all the characters of the original font and set the character width back to 9 pixels for VGA systems. Both LOGO demonstration programs use the BuildLogo function to create a small logo and display it on the screen. The ASCII character set is displayed in the portion of the screen above the logo. The characters that were redefined to form the logo will be highlighted with a different color. This allows you to see each redefined character and how they fit together like the pieces of a puzzle to form the logo.

You'll find the following program(s) on the companion CD-ROM



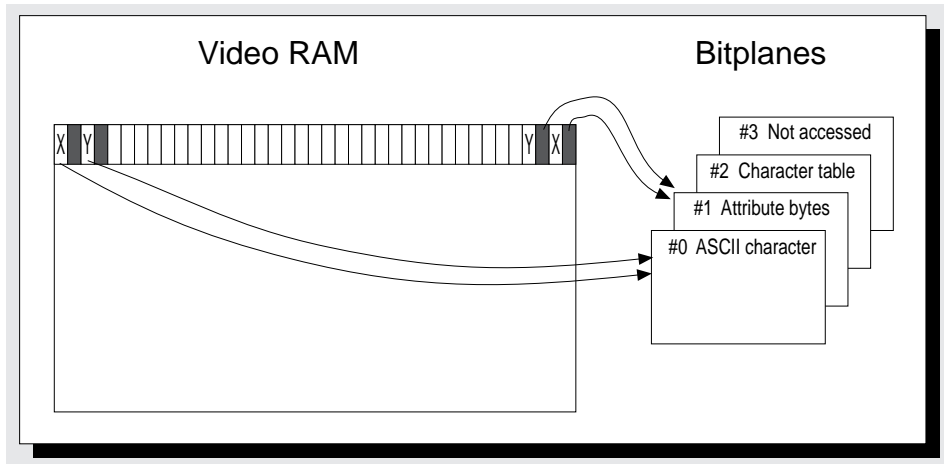
LOGOP.PAS (Pascal listing)
LOGOC.C (C listing)
LOGOCA.ASM
(Assembler listing)

The Pascal version LOGOP.PAS doesn't require any assembler routines, but the LOGOCA.ASM assembly language module supports the C version LOGOC.C. This assembly language module supplies character definition information because the BIOS function that defines characters cannot access character information in the BP register using the int86() C input function. The defchar routine in the assembler module handles character definition.

Character table structure and location

You don't need to worry about the structure and location of character tables as long as you use only the BIOS to access them. However, occasionally you'll have to access them directly. Let's examine when this occurs and see how to access character definitions directly from character tables by using BIOS functions.

*Using bitplanes
in text mode*



We'll begin by studying the structure of the video RAM on EGA and VGA cards. In "Understanding bitplanes" later in this section we note that EGA and VGA cards divide their video RAM into four large areas called bitplanes. These areas serve different purposes in text and graphics modes. A fully populated video card (with 256K) allocates 64K per bitplane.

The first two bitplanes (0 and 1) contain character codes and attribute bytes in text mode. The ASCII character codes are stored in bitplane 0. Accessing the video RAM above B800H at an even offset address is directed to this bitplane. Accessing the video RAM at an odd offset address is directed to bitplane 1, which contains the attribute bytes. The other bitplanes are also used in text mode. EGA and VGA cards normally use the third bitplane for the character table. In a fully populated video card, each bitplane has 64K of RAM. Since each character table requires 8K, this allows you to store 8 different character tables. However, only the VGA card uses this bitplane fully. Since the EGA card uses only four character tables, 32K of memory remains unused.

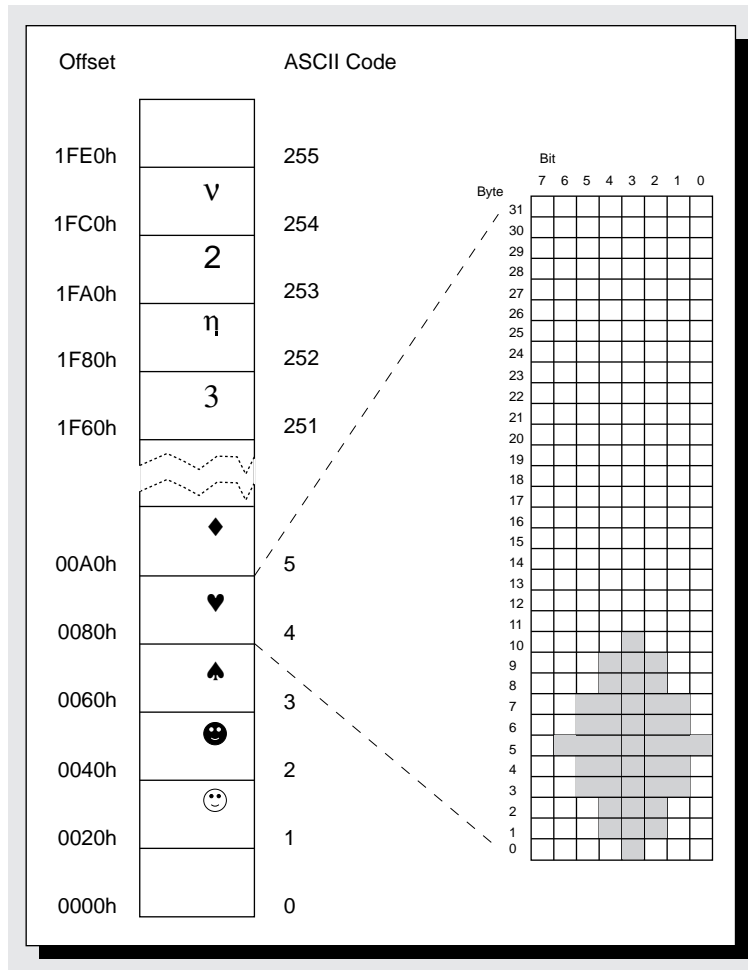
*Location of
character tables
within bitplane 2*

EGA	Offset	VGA
Unused	E000h	Character table #7
Character table #3	C000h	Character table #3
Unused	A000h	Character table #6
Character table #2	8000h	Character table #2
Unused	6000h	Character table #5
Character table #1	4000h	Character table #1
Unused	2000h	Character table #4
Character table #0	0000h	Character table #0

Each character table needs 8K because 32 bytes are reserved for each of the 256 characters ($32 \times 256 = 8192$ bytes or 8K). Remember, 32 pixels is also the maximum character height for EGA and VGA fonts. The character tables are organized for the maximum font size. If a smaller font is used, then the extra bytes at the end of each entry simply remain unused.

The bytes that are used are coded in connection with the BIOS functions for character definitions. Each byte represents the bit pattern for one pixel line of the character. The individual bits in each byte indicate whether the corresponding pixel is on or off. You can easily determine the starting address for a specific character or a pixel line within a character if you know the starting address for the character table. Simply multiply the ASCII code number for the character by 32, then add the starting address for the table and the number of the desired pixel line.

Character table structure



Before accessing a character table, we must perform one more step. Since direct access to bitplane 2 isn't allowed, its contents must first be loaded into memory, from which it can be accessed from segment address A000H. Unfortunately, a BIOS function doesn't do this. So, again we must program the EGA and VGA registers.

Specifically, we'll be working with various registers of the sequencer and graphics controllers. Registers 2 and 4 of the sequencer controller; are known as the map mask register; and the memory mode register. The first register determines which bitplane is accessed. Each bit in this register represents a bitplane. In this case, we want bit 2 to be set for bitplane number 2. All other bits should be cleared. This ensures that only bitplane 2 is accessed.

To ensure that only the bitplanes indicated in the map mask register are accessed, you must also load the value 7 in the memory mode register. In normal text modes, this register contains the value 3. This means that when the B800H video RAM is accessed, all even memory addresses (ASCII codes) go to bitplane 0 and all odd memory addresses (attribute codes) go to bitplane 1. Changing the value to 7 changes the memory addressing method.

We must also reset the sequencer registers by accessing the reset register, like we did when changing from a 9 pixel to an 8 pixel character width. So before accessing the map mask and memory mode registers, we load the value 1 and the value 3 in the reset register. A reset isn't required to access the graphics controller registers. We need to manipulate registers 4, 5 and 6 to access bitplane 2. These are called the read map select register, the graphics mode register and the miscellaneous register.

The read map select register; is loaded with the value of the bitplane we want to read (2, in this case). This means that read access to the video RAM is also directed to bitplane 2. The graphics mode register requires a value of 0. This is accomplished by simply clearing bit 4, since all other bits pertain only to graphics mode. Clearing this bit ensures the odd and even memory addresses won't be split into different bitplanes.

This information is repeated in bit 1 of the miscellaneous register. We're also interested in bits 2 and 3 of this register. These bits indicate where we can find the video RAM and where the bitplanes should be stored. In text mode, the video RAM is located at segment address B800H and extends for 32K. Bitplane 2 begins at segment address A000H, allowing access to the entire 64K. Setting up the registers in this way destroys the normal text mode settings. So, now you're able to access bitplane 2, but you can no longer access the normal video RAM at B800H. Any character output that occurs during access to bitplane 2 is ignored by the video RAM. The picture on your screen appears unchanged because the video card can directly access the video RAM internally.

Remember, for your program or the BIOS to access the video RAM again, you must reset of the registers to their original values. The table on the left shows the proper register values for access to bitplane 2 and for access to video RAM at B800H.

Register	Bitplane	Video RAM
Map mask register (sequencer controller)	04H	03H
Memory mode register (sequencer controller)	07H	03H
Read map select register (graphics controller)	02H	00H
Graphics mode register (graphics controller)	00H	10H
Miscellaneous registers (graphics controller)	04H	0EH

The demonstration program MIKADO (listed later in this chapter) contains

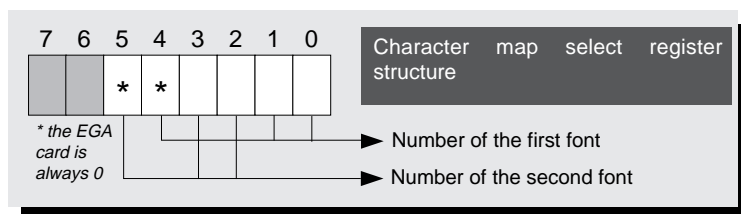
two routines used for toggling access between video RAM and bitplane 2. Also, at the end of this chapter, you'll find additional information about the registers we just discussed.

Switching between fonts: 512 different characters simultaneously

We've seen that bitplane 2 can store four different fonts on an EGA card and eight different fonts on a VGA card and that BIOS access isn't limited to the first font. Now we must determine how to switch between fonts to display the desired one on the screen. We must use register number 4 (the character map select register) of the sequencer controller. This register is responsible for selecting the current character table, as the illustration below shows.

As you can see, the character map select register works with two fonts instead of only one. The order of the bits that specify the two fonts is confusing. Instead of using three consecutive bits to identify the font, each font has a group of two bits and one additional bit that's separate from the other two. The single bit is the highest bit of the group. The reason for this can be traced back to the development of EGA and VGA cards. Since EGA cards have only four fonts, they only need to represent the numbers 0 through 3, which can be done with 2 bits.

VGA cards have eight available fonts, so we need to represent the values 0 through 7. To maintain compatibility, the VGA card was developed to imitate the original two bit configuration of the EGA card. A single



extra bit was then added to accommodate the extra fonts. Since this extra bit was never used in the EGA card, it doesn't interfere with its operations.

The fonts listed in the previous figure are called first font and second font. However, this doesn't represent a hierarchical relationship. Instead, this is what gives the EGA and VGA cards the ability to display 512 characters on the screen simultaneously, unlike their predecessors, which can display only 256 characters. You may be wondering how to select characters from one of these fonts. Since there are only 256 ASCII codes, this isn't the answer.

At first, you might think the attribute byte for a character doesn't have enough space for this kind of information. The two nibbles of this byte select a background and a foreground color for the character. However, this is the key to working with 512 characters simultaneously.

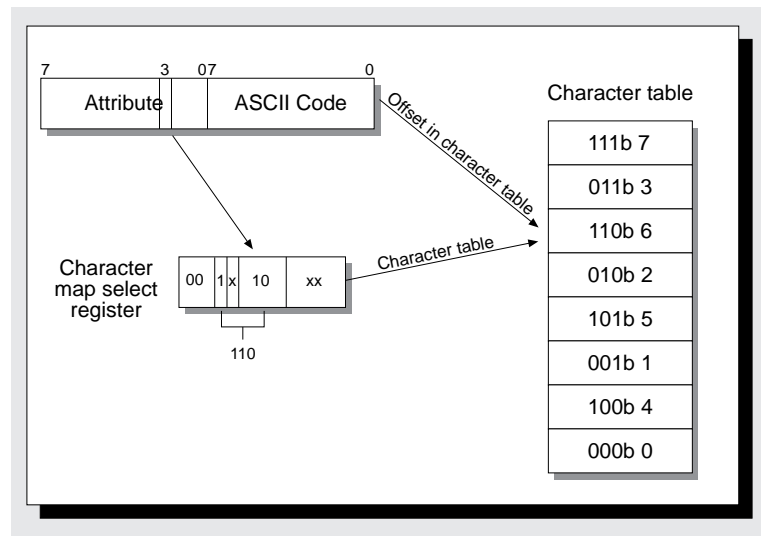
The highest bit of the foreground color (bit 3 in the attribute byte) is used to select the first or second font. If this bit is set to 0, then the foreground color is less than 8 and the video card will use the corresponding character from the first font. If this bit is set, then a foreground color greater than or equal to 8 is indicated and the character is taken from the second font.

Although the user may not notice, a distinction is always made between the first and second font. This is because the same font is specified twice in the character map select register. Once two numbers are entered in this register, the difference between the fonts immediately becomes apparent on the screen.

Access this register if you want to display two fonts on the screen simultaneously or if you just want to switch to another font. You can produce some very interesting visual effects by simply switching between two fonts in your application.

You don't have to program the character map select register directly because the video BIOS uses sub-function 03H of function 11H to do this. This sub-function expects the two function numbers in the AH and the AL registers. When sub-function 03H is called, an additional parameter must be passed to the BL register. This parameter is the value that you want to load in the character map select register to indicate a desired font. The following section contains an example of how this function is used. This example uses the second font to create a graphic window within a text screen.

Selecting fonts from the character map select register and attribute byte



One problem you may encounter when using a second font is the characters of the second font may appear lighter on screen. This occurs because bit 3 of the attribute byte is set, which results in a foreground color number of 8 or greater being used

for characters of the second font. This problem can easily be solved by simply changing the palette registers to set foreground colors 8-15 equal to those assigned to 0-7.

Since the BIOS has many functions for handling this type of manipulation, this operation can easily be performed. Although this limits the number of foreground colors to eight, few programs would use that many foreground colors simultaneously. You could program the palette registers yourself to obtain additional colors if necessary.

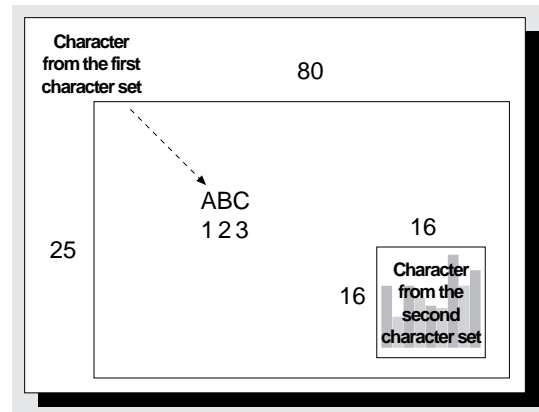
A graphics window in text mode

Suppose that instead of displaying 512 characters, you need mathematical symbols or other special characters. There's an alternative to using the character select map register. Instead, you can use the second font to create a graphics window in your text screen. Then you can display small graphics or images without switching to the more complicated graphics mode.

The technique for creating a graphics window is similar to the technique for creating a logo. Simply piece together a number of specially defined characters to form the desired image. Each character from the second font appears once in the graphics window. This is important because it limits the size of the graphics window to 256 characters. The resulting block may be as large as 16x16 characters, but you aren't limited to any particular shape of window.

The second font is considered an array of individual pixels relating to coordinates within the graphics window. To set or clear a pixel within the graphics window, first you must calculate which character contains the appropriate pixel. The pixel position is determined relative to the upper-left corner of the character. The pixel is then set or cleared in the bit pattern for the character in the character table.

Graphics window structure in text mode



For example, suppose that a VGA character comprises 16 pixel lines, each consisting of eight pixels. If we take the lower-left corner of the screen as the point of origin for our graphics window, then all pixels with X-coordinates less than 8 and Y-coordinates less than 16 are considered part of the first character, which is in the lower-left corner of the graphics window. As the previous figure indicates, this character is always ASCII code 0. The number of each character in our window remains fixed.

The next step calculates which pixel line within the character we want to manipulate. This structure is the same as that found in the character table. So, a pixel with a Y-coordinate of 15 would be in the top pixel line of the character with ASCII code 0 and a pixel with a Y-coordinate of 0 would be in the bottom line of the character. This enables us to easily calculate the location of any pixel.

The final step determines the location of the desired pixel within the pixel line. The bit positions of the pixels are numbered 7 to 0 from left to right. By knowing the character number, pixel line and bit number, any pixel can be switched on or off by passing the corresponding information to the character table in video RAM.

Redefining an entire character every time you want to set or clear a single pixel may seem inefficient. However, if you're using the BIOS functions for defining characters, there's no alternative. Also, since these functions aren't very fast, you'll notice

problems when you want to process pixels in rapid succession (e.g., when drawing a line). For these reasons, both programs presented in this section access bitplane 2 directly. This allows you to manipulate the graphics window effectively.

Unfortunately, there is a problem with VGA because of the ninth pixel. We've already encountered this problem while using the LOGO program. As you may remember, the ninth pixel always remains blank, so we must limit the VGA character width to 8 pixels so the characters will flow together smoothly and without an empty space between them. We described the procedure for changing VGA character width from 9 pixels to 8 pixels earlier in this chapter.

The demonstration program discussed in this section is called MIKADO. Again, we've included both Pascal (MIKADO.PAS) and C (MIKADO.C) versions. The highlight of these programs is a routine called InitGraphArea routine, which configures the graphics window. The programs also include a routine called SetPixel, which allows you to set or delete individual pixels. This routine acts as the basis of the Line function, which draws a line within the graphics window.

The InitGraphArea procedure accepts data about the location of the graphics window, its size and the colors to be used. You can also select the font number that you want to use to create the graphics window. The maximum X and Y coordinates are calculated from the character resolution and the window's size. These values are then stored in the xmax and ymax global variables.

Bitplane 2 is accessed using the procedure we described (in the GetFontAccess routine). The ReleaseFontAccess function performs the opposite task; it restores access to the video RAM at B800H. These routines aren't called very often because "snow" appears on your screen while they are executing.

The SelectMaps procedure is also important to the MIKADO program. This routine enters the desired fonts in the character map select register, using the BIOS function we described earlier. This prevents direct hardware access to the video card. Bypassing hardware access makes the program more compatible with all types of VGA cards because different manufacturers may assign different uses to the registers.

Several procedures in the MIKADO programs use the BIOS to program the palette registers. Any element of the 16 color palette can be changed, the entire palette can be redefined or the current contents of the palette can be queried. The procedures that set the colors for the text and graphics windows are called SetPalCol, SetPalAry, GetPalCol and GelPalAry.

When considering colors for the graphics window, remember that color must be assigned by character instead of by pixel, which is possible in graphics mode. A character consists of 14x8 pixels with EGA and 16x8 pixels with VGA. Usually it's best to assign one background color and one foreground color to the graphics window. However, the MIKADO programs do something entirely different. These programs assign a successive color code to each character in the graphics window. This gives the illusion of gradually changing bands of color.

These two programs also clarify the order in which the various routines must be called. First, IsEgaVga determines whether an EGA card or VGA card is available. Only then can the graphics window be opened with InitGraphArea. Before the window can be accessed, GetFontAccess must be called. Then you can draw lines or pictures within the window using SetPixel.

If you want to send text to the screen, you must call ReleaseFontAccess first, then call GetFontAccess again to return to graphics output. ClearGraphArea clears the graphics window, then restores access to video RAM and the default font (font 0). MIKADO fills the screen around the graphics window with ASCII characters, showing the complete ASCII character set is still available with the graphics window in place.

The MIKADO programs contain of the tools you'll need to work with a graphics window in text mode. After studying the listing, you'll be able to modify or expand these routines according to your own needs. Having a graphics window open in the middle of a text screen can be very useful. Now you can add this feature to your own applications.

You'll find the following program(s) on the companion CD-ROM



MIKADOP.PAS (Pascal listing)
MIKADOC.C (C listing)

Fonts for graphics mode

Character tables also display characters in EGA and VGA graphics modes by using the corresponding BIOS functions. Since it's already being used to store pixel information for the graphics screen, bitplane 2 won't be available for storing character tables in graphics mode. So, the pixel pattern for each character must be passed to the BIOS. This can be done either directly from the ROM chip or from a RAM buffer. However, you must determine which font to access.

If you don't tell the BIOS which font you want to use, it defaults to the font enabled for 80x25 character text mode. EGA cards default to the 8x14 font and VGA cards default to the 8x16 font. To determine the number of text lines that will fit on the graphics screen, divide the vertical resolution by the pixel height of a character. Dividing the character width (8) by the horizontal resolution determines the number of characters per line.

You don't have to use the default font for text output in graphics mode. The BIOS allows you to access any font available in ROM. To do this, use sub-functions 22H, 23H and 24H of video BIOS function 11H. The system documentation indicates that in addition to the function numbers in the AL and AH registers, you must also pass two other items of information in the BL and DL registers. Based on our experiences with the original IBM cards, the information passed in these registers doesn't affect text output in graphics mode and can be ignored. You only need the function numbers to call these functions.

You can also use sub-function 21H to load your own character table for graphics mode. This character table must include all 256 characters. This call requires the function numbers in the AH and AL registers and the individual character height in the CX register. As with all other BIOS functions used to define characters, this function requires a pointer to the character table in the ES:BP register pair. The structure of this pointer corresponds to that of the character table pointers used by sub-functions 00H and 10H.

Sub-function.	Matrix	EGA	VGA
22H	8x14		
23H	8x8		
24H	8x16	(n.a.)	

Smooth scrolling

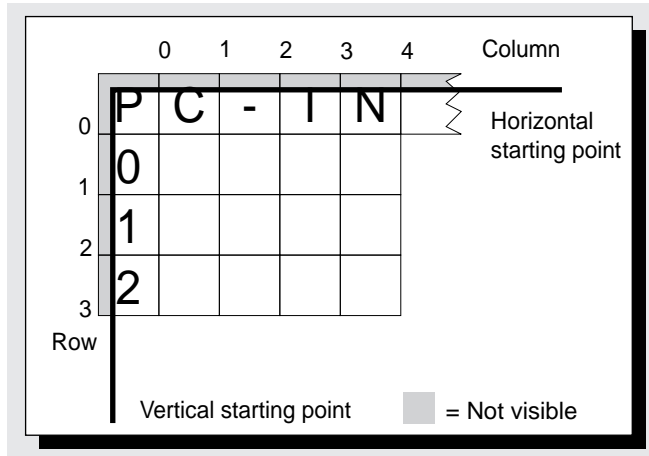
Prior to EGA and VGA, it was impossible to create a smoothly moving picture with video cards. It was possible to scroll one character at a time in a single direction in text mode. But to scroll in graphics mode, you had to recopy the entire video RAM, even if you only wanted to move the screen by one pixel. This was one of the reasons why the PC lagged so far behind other computers in pixel animation.

EGA and VGA cards offer some relief. The video hardware on these cards contains the ability to move the screen pixel by pixel and perform smooth scrolling. This made the first convincing animated graphics possible on the PC and opened new horizons in text mode.

Smooth scrolling using the pel panning register

In the EGA and VGA cards, two registers are used to move the screen display pixel by pixel. These two registers control different parts of the video hardware, but they work together to perform basically the same task, which is setting the screen origin (the horizontal and vertical starting points of the screen). The vertical pel panning register is part of the CRT controller and the horizontal pel panning register is part of the attribute controller.

The horizontal and vertical starting points of the screen



Let's first look at the horizontal pel panning register. In its normal state, this register contains a value of either 0 or 8. These values indicate the character width in text mode, ensuring the correct horizontal proportions in characters. A value of 8 is the default setting for all video modes with a character width of nine pixels. This applies to the VGA card and the EGA card when emulating an MDA card (i.e., when the EGA is attached to a monochrome monitor). The default value for an EGA card attached to a color monitor is 0.

Incrementing the value in the horizontal pel panning register moves the contents of the entire screen to the left, pixel by pixel. For an EGA card with color monitor, the values would be 1, 2, 3, etc. The larger the number, the further to the left the screen scrolls. The maximum value is 7, which scrolls the screen the width of one entire character. You cannot use the horizontal pel panning register to scroll more than the width of one character.

This procedure works differently for VGA cards and EGA cards in MDA emulation mode. As we mentioned, the default value is 8. The numbers 0 to 7 perform scrolling to the right. Unlike left scrolling, you must begin with the maximum number (7) and decrement the value to 0. With VGA cards and EGA cards with MDA monitors, you can skip up to a value of 8 for the final step.

The horizontal pel panning register can be applied to text mode and graphics mode. There are two different cases to consider in graphics mode: The VGA card's 256 color mode and all other modes. In 256 color mode, the starting point is 0. You can scroll the screen up to three pixels to the left, instead of entering 1, 2 or 3 as you might expect. However, using the horizontal pel panning register in graphics mode requires values of 2, 4 or 6 to scroll.

All other graphics modes also have a starting point of 0. The horizontal pel panning register will accept values from 1 to 7 to scroll the screen a maximum of seven pixels to the left.

You can load the horizontal pel panning register either by programming the attribute controller directly or by using sub-function 00H of BIOS video interrupt function 10H. This BIOS function was originally intended for loading a specific palette register. However, since the pel panning register is found in the attribute controller just like the palette registers, it can also be accessed using this function. Enter the number of the horizontal pel panning register instead of one of the palette registers. When calling this function, enter the two function numbers as usual. Then pass the number of the horizontal pel panning register (13H) to the BL register and the value you want to pass to the BH register.

If speed isn't an important requirement, you should use this method instead of accessing the attribute controller. To access the attribute controller directly, your program should be able to distinguish between EGA and VGA cards. However, since most manufacturers of EGA and VGA cards have adhered to the standards for assigning registers, you shouldn't have any problems if you want to program the horizontal pel panning register directly.

If you prefer the direct route, first the number of the horizontal pel panning register (13H) must be passed to the combined data/index register of the attribute controller at port address 3C0H. When doing this, remember to set bit 5 in this register.

This bit enables and disables the attribute controller. If this bit isn't set, the attribute controller remains disabled and the screen will be black.

After sending value 33H (register number 13H plus bit 5) to port 3C0H, you can pass the new value that should be placed in the horizontal pel panning register. This value is the pixel counter that controls the degree of screen scrolling.

Vertical scrolling

Vertical scrolling using the vertical pel panning register is less complicated because the starting point is always 0, regardless of the mode or video card type. Any value greater than 0 scrolls the screen the corresponding number of pixels up and any value less than 0 scrolls the screen down. In the normal 80x25 character text mode, the maximum values for vertical scrolling are 13 for EGA and 15 for VGA.

In graphics mode, the vertical pel panning register accepts values from 0 to 31, where 31 (not 0) is the starting point. Any smaller value moves the screen the corresponding number of pixels down. If you want to scroll up, set the starting point from 31 to the minimum scroll value, then increment that value.

To scroll down in text mode, the procedure is the same as horizontal scrolling to the right. Start with the highest value and decrement the value until you reach the starting point. The vertical pel panning register must be accessed directly using the CRT controller; there are no BIOS functions available. First pass vertical pel panning register number (08H) to the CRT controller's index register.

The port address of this index register depends on the video card's current operating mode. In color mode, the port address is 3D4H. In monochrome mode, it is 3B4H. The data register, which immediately follows the index register, must be loaded with the new value for the vertical pel panning register. Instead of two 8-bit operations, you can perform a single 16-bit operation and load both the index and data registers simultaneously.

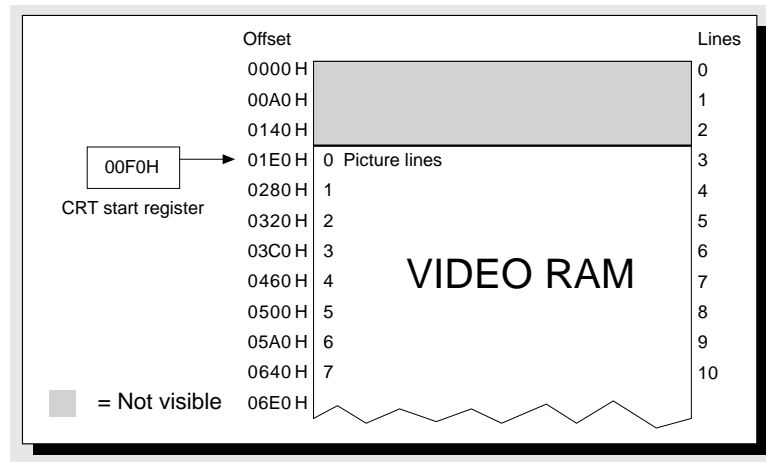
In practice, the pel panning registers are rarely used for scrolling the screen. A developer usually wants to scroll continuously instead of one character at a time. You could scroll the screen in this way, reset the pel panning registers to their starting points, move the entire contents of video RAM and repeat the procedure to obtain continuous scrolling. However, moving the characters in video RAM takes too much time and results in a slower execution time.

The technique described in the following paragraphs is usually preferable. It involves moving the screen display's point of origin and changing the line length in video RAM.

Moving the screen origin and changing the line length in video RAM

One way to scroll the screen contents up by one text line is to move the contents of the video RAM up by 160 bytes (160 bytes is equivalent to one line in 80x25 character text mode). You could also simply increment the starting address of the current screen page by 160 bytes. The CRT controller will begin the display with what was originally the second line when the screen is repainted. Regardless of which method you use, the result is the same to the user.

Instead of moving everything in the video RAM, it's much easier, from a programmer's point of view, to change the contents of the CRT controller registers and move the starting address of the current screen page. This same register also allows you to jump directly from one screen page to another. This method also stores the lines you've scrolled past in video RAM so they can be recalled later.

Horizontal scrolling

Changing the starting address of the video RAM is similar to using your arrow keys to scroll to different pages of a document in a word processing program. Instead of using arrow keys, we use the CRT starting registers. And instead of the text in a document, we're scrolling the contents of the video RAM.

The numbers of the two registers we use are 0CH and 0DH. These registers contain the starting address of the portion of the video RAM that's currently visible on the screen. These registers cannot be reached directly using the BIOS, so again we must program the CRT controller directly. You should remember two important points when doing this. First, the starting address is given in the form of a 16-bit offset address with its high byte in register 0CH and its low byte in register 0DH. The order is reversed from what you would normally expect because the high byte appears before the low byte. Also, this offset is counted in words rather than bytes, so you must divide by two before writing the address to the register.

You can easily scroll vertically by accessing the pel panning register and changing the screen origin. However, this method has some unpleasant surprises if you try to use it to scroll horizontally. For example, suppose that you scroll one character to the left and then increment the starting address from 0000H to 0002H. The character from the second column of the first line is now in the first column. But the first character from the second line now appears in the last column of the first line and so on down the screen. This is known as character wrapping.

Character wrapping when moving the screen origin

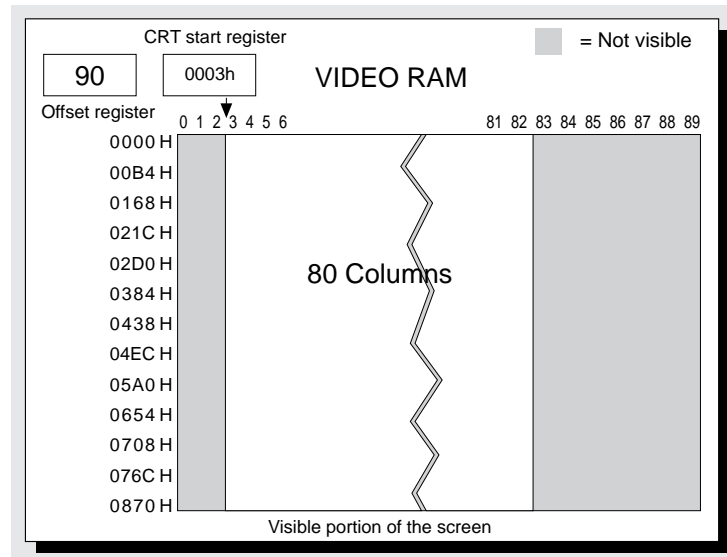
The character wrapping problem is related to video RAM organization; this problem can't be solved using the CRT start registers. Fortunately, we don't have to move the entire contents of the video RAM. The CRT controller has another register called the offset register. This register stores the length of a text line in video RAM. The normal value for this register is 80 words, which corresponds to a line length of 160 bytes.

Increasing this value doesn't instruct your system to display more characters per line on the screen. Instead, the CRT controller's internal address counter increments to make more room per line in the video RAM. For example, if you placed a value of 82 (52H) in this register instead of the normal value of 80 (50H), then each line in video RAM would contain 82 characters. Only the first 80 are visible on the screen until you scroll. Once you scroll, the last two characters in the line become visible and the first two characters move beyond the left border of the screen.

The offset register (register 13H in the CRT controller) is 8 bits wide. So, it can accommodate line lengths of up to 255 characters, which gives each line a length of 510 bytes in video RAM.

Remember this when you're addressing a certain line within the video RAM. Before, you would always multiply by 160 to access a certain line. Now, you must multiply by 510, 320 or whatever the video RAM line length is at the time. In all cases, this number will be twice the number of characters per video RAM line.

Scrolling the screen by increasing line length in the offset register



Once you've increased the internal line length using the offset register, you can easily scroll the screen by using the pel panning register and changing the screen origin. Another important point to consider when scrolling (both vertically and horizontally) is the synchronization of events. If you aren't careful with synchronization, the result on the screen may be completely different from what you had intended.

Synchronization with the CRT controller

Scrolling the screen by manipulating the pel panning register and screen origin must always be coordinated with the CRT controller. First, the pel panning register should be accessed only during vertical synchronization of the electron beam (i.e., when visual information isn't being sent to the screen). This ensures the changes to these registers won't affect any parts of the screen that are created with different values in these registers.

If the pel panning register; and the CRT start register must be programmed simultaneously (e.g., to return the pel panning register to its starting value and to move the screen origin), then you must be very careful. While the changes to the pel panning register are considered immediately as the next screen is built, the change to the CRT start register takes longer to process.

This occurs because a program usually hesitates after starting vertical synchronization, even if the hesitation is only a fraction of a second. The hesitation occurs because of querying the vertical synchronization. This query is usually done using bit 3 of the input status register, which shows the vertical synchronization status. It's on during synchronization and off at all other times. The query takes place in a program loop that constantly checks the status and the hesitation occurs because the assembly language instructions in this query need some time to execute. The delay is worse if you program in a high level language, in which the query code will be even slower.

Once the program recognizes the vertical synchronization, it may already be a fraction of a second into its execution and the CRT controller may have already loaded the starting address for the next screen refresh from the CRT start register. So, any change to the contents of this register won't be considered until the subsequent screen refresh.

Changes to the pel panning register take effect with the next screen refresh. This results in internal inconsistencies that are visible on screen. To avoid these problems, you must follow a specific procedure when programming the pel panning register and the CRT start register simultaneously.

This procedure uses input status register 1 to wait for a vertical synchronization to start and finish. Then the new screen origin is loaded in the proper register of the CRT controller. This won't disturb the next screen refresh because the CRT controller has already loaded the screen origin address into its internal address counter before the screen refresh and doesn't have to access the register again.

Now, wait for the screen to regenerate and for the next vertical synchronization. After this event, you can program the pel panning register. The change to this register is then taken into account with the next screen refresh. Now we can be sure the address of the new screen origin is correctly loaded into the internal address counter for this screen refresh as well.

Algorithms for setting the pel panning register and CRT start register

Waiting for the vertical synchronization also synchronizes your program with the video frequency of the video card, regardless of the CPU speed. This is very useful, especially with games. If you continuously call the routine for setting the pel panning register and the CRT start register, you can move the screen contents by a few pixels each time the picture regenerates. This involves 50 to 70 movements per second, depending on whether you have an EGA or VGA card. Since the human eye cannot follow so many rapid changes, it interprets this as smooth animation.

Scrolling in text mode

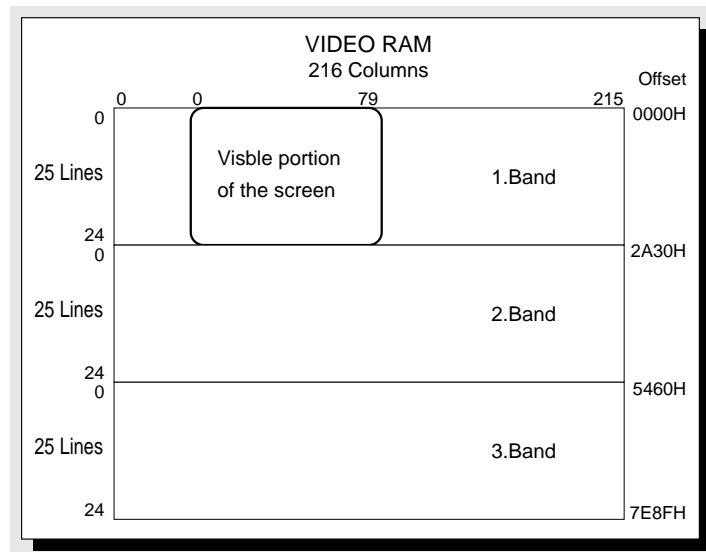
The SOFTSCR.PAS and SOFTSCRC.C programs which you'll find on the companion CD-ROM are examples of this combined control of the pel panning register and the CRT start register. Each program scrolls text printed in a large font, from right to left across the screen.

These programs are based on two routines called ShowScrText and SetOrigin. ShowScrText takes the text to be displayed and stores it in the video RAM so it's not completely visible at once. The program divides the text into three sections, each 216 characters long by 25 characters high. Each section uses 10,800 bytes (or a total of 32,400 bytes), leaving only 368 bytes of video RAM unused.

After the text is placed in video RAM, ShowScrText begins the display. The SetOrigin procedure is important because it sets the starting point for the screen. The section of text, the starting row and column and the number of pixels to move must be specified. The number of pixels is entered directly in the pel panning register. The number of the text section and the starting column and row are linked to a starting address of the visible screen and loaded in the CRT start register.

First SetOrigin is called from within ShowScrText using constant values for the starting column and row. Only the pixel counter increments to create the horizontal movement. Then when the maximum value for the horizontal pixel counter is reached, the counter resets to 0 or 8 and the starting column increments.

Starting addresses of the interrupt routines are arranged in the interrupt vector table



The SOFTSCR.PAS and SOFTSCRC.C programs allow the user to control the scrolling speed. The differences in character width between EGA and VGA are also taken into account. To control this, the ShowScrText routine accepts two parameters.

One parameter is used for the scrolling speed and the other is used for the video card type. Speed is determined by selecting either the FAST, MEDIUM or SLOW constant and the video card type is determined by entering either EGA or VGA.

While the screen origin is being moved in ShowScrIText, these two constants are converted to an index in the array step table. This table is declared within the routine. Depending on the speed and video card type, a series of values for the horizontal pixel counter is selected. The value 255 indicates the end of each row of values in the array, which means that scrolling continues in the next column.

Different scrolling speeds are obtained with different intervals in the value of the horizontal pixel counter. In SLOW mode, each pixel in each character is scrolled. In MEDIUM mode, every other pixel is skipped, which doubles the scrolling speed. The speed is doubled again for FAST mode, in which only two pixels in each character are scrolled.

The value 216 indicates the limit for incrementing the starting column. This value refers to the last column in a section. The program must switch to the next section of text and start again with column one. But since the contents of the last screen in the previous text section can no longer be scrolled left off the screen, the next section must begin with the same characters found at the end of the previous section. This allows you to program a completely smooth transition between sections of text.

The speed at which the user can comfortably read the characters that are scrolling by depends mostly on the character width. The PrintChar routine; is responsible for building the characters in video RAM. The arguments PrintChar requires are the character code and the column number and text section number where the character should appear. The row number is given by the constant STARTR, so it doesn't have to be passed again. The routine obtains the bit pattern for the characters from the 8x14 pixel font, which is included in ROM on both EGA and VGA cards. This font is accessed using sub-function 30H of BIOS video interrupt function 11H.

This routine is also the reason why the C version of this program requires the small assembler routine called SOFTSCCA.ASM. The BIOS function mentioned returns information to the BP register, which cannot be accessed by normal C interrupt functions.

PrintChar sends the 8x14 character matrix to the screen by mapping each pixel in the pattern to a character position on the screen. This means that one character will occupy 8 screen columns and 14 screen rows. So, one section of the text contains 27 characters (216/8). Any 10 characters (80/8) are visible at a time. However, remember the second and third sections of the text must repeat the last screen of the previous section. This means that sections 2 and 3 can only contain 17 new characters instead of 27. Therefore, the total number of characters that can be loaded in video RAM is 61 (27 + 17 + 17).

You'll find the following program(s) on the companion CD-ROM



SOFTSCR.PAS (Pascal listing)
SOFTSCRC.C (C listing)
SOFTSCCA.ASM (Assembler listing)

Disabling the screen

Occasionally the screen display of your video card must be disabled. Screen saver programs do this to prevent the screen image from burning into your monitor. EGA:Disabling screen; The EGA and VGA cards offer capabilities for disabling screen display, which we'll discuss in this section. Specifically, we'll look at the attribute controller, its role in generating the video picture and how this can be used to disable the screen display.

The attribute controller's role

The four basic control components in EGA and VGA cards are the CRT controller, the graphics controller, the attribute controller and the sequencer controller. The attribute controller assigns color information to the picture. If its operation is interrupted, then no color information will reach the screen and the result is a black screen.

It's much easier to interrupt the attribute controller than the other controllers found on EGA and VGA cards. When bit number 5 in the index register is set to 0, the attribute controller is switched off and color information doesn't reach the screen.

This bit is only significant to the attribute controller. To the other controllers, the index register only accepts the number of the register being addressed. The reason for the difference in the attribute controller involves the palette register (the attribute registers most frequently addressed). Before accessing a palette register, the attribute controller must be temporarily stopped. Since accessing a palette register requires access to an index register anyway, it makes sense to put the bit used for suspending the attribute controller in the index register also. And since there are only 21 registers available to the attribute controller, only bits 0 to 4 of the index register are needed for accepting a register number.

However, before we can disable the attribute controller by clearing bit 5, we must first read the CRT controller's status register. This makes the attribute controller reset effective and must be done before enabling or disabling the attribute controller.

Sample programs

The VONOFFP.PAS and VONOFFC.C programs demonstrate how to enable or disable the screen display using the attribute controller. First, these programs check to ensure the system is using an EGA or VGA card. If it isn't, an error message is displayed and the program ends. If an EGA or VGA card is found, a different message is displayed. The user is given five seconds to read this message, then the ScrOff routine disables the screen display.

The ScrOff routine starts by reading the status register of the CRT controller. To make this routine work with both color and monochrome monitors, a double reset occurs: Once using the monochrome address of the status register (3BAH) and once using the color address (3DAH). This won't cause any problems, since only one of these two ports will be used.

Next, the value 0 is written to the attribute controller's index register at port address 3C0H, bit 5 is set to 0 and attribute controller activity is suspended. The screen display goes black. The ScrOn routine executes after the user has pressed a key. This routine writes the value 20H to the attribute controller's index register. This activates the controller again and enables screen display.

You'll find the following program(s) on the companion CD-ROM



VONOFFP.PAS (Pascal listing)
VONOFFC.C (C listing)

Understanding bitplanes

The technological progress of PC video cards has produced higher resolutions and increasing numbers of colors. As resolution increases, so does the amount of video RAM required. As the need to display more colors increases, more than one bit is needed to represent each pixel. For example, VGA cards with 256 colors require 8 bits per pixel. As a result, almost all EGA and VGA cards manufactured today include 256K of RAM as standard equipment. Depending on the video mode and the number of screen pages stored, this memory is often used to capacity.

Although the 64K limitation of early video cards no longer exists, we can't continue to add more video RAM. Remember, PCs are limited to 1 megabyte of addressable memory. In this section, we'll discuss how the PC's addressable memory handles the 256K of video RAM and how this affects programming EGA and VGA cards.

Dividing video RAM into bitplanes

In the total addressable memory of the PC, only segments A (from A000:0000) and B (from B000:0000) are available to the video card. Segment B is already reserved for the video RAM in MDA, CGA and Hercules cards. So, only segment A can be used for EGA and VGA cards. This means the developers of the first EGA card had to address 256K of video RAM in a 64K area.

The solution resulted in the development of bitplanes, which divide the video RAM of EGA and VGA cards into equal sections. On a card with 256K of video RAM, each bitplane has 64K that is completely addressable using memory segment A, starting at A000:0000. If your video card has less video RAM, such as the first EGA cards that had only 64K, then each bitplane is correspondingly smaller.

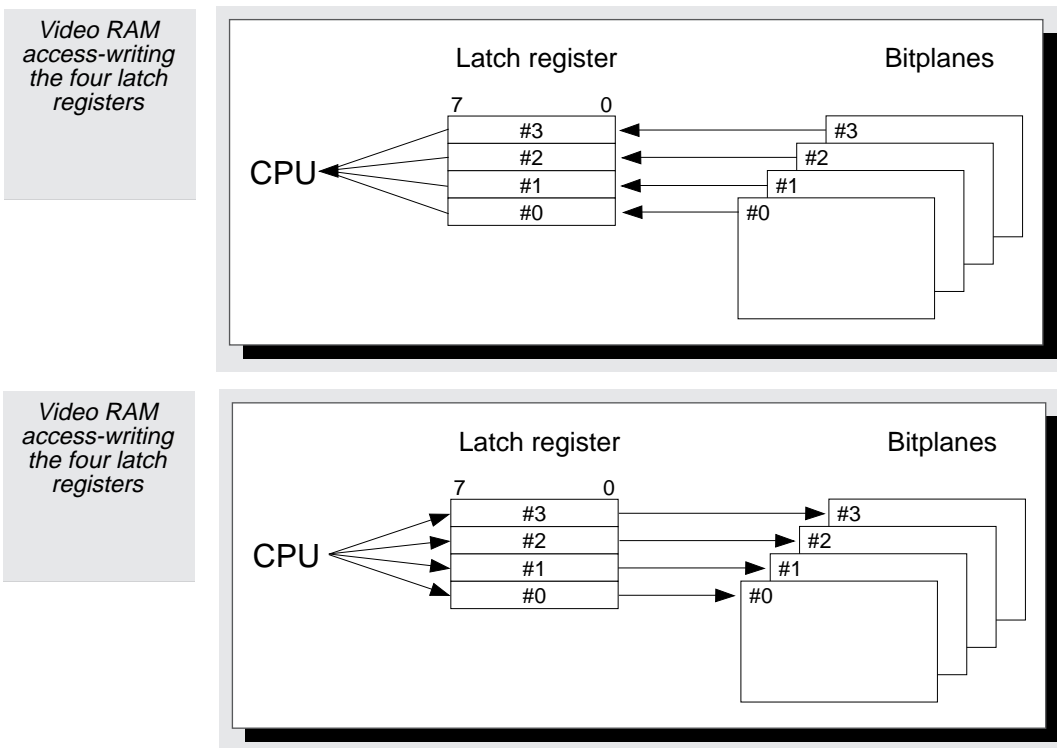
This memory addressing technique, which was developed in 1984, is valid for all EGA and VGA cards. This includes the Super VGA card, although the addressing is used in a slightly expanded form (more on this later).

Latch registers

Use different bitplanes in text modes only if you want to use more than one font simultaneously. Bitplanes are always used when graphics modes are active. Pixel information is spread out among the bitplanes in various ways for different modes. Regardless of this, EGA and VGA cards manage communications between the processor and video RAM using four 8-bit registers known as the latch registers. Each latch register corresponds to one of the four bitplanes.

Latch registers cannot be directly accessed by programs. If a program wants to access a byte in video RAM, then the four latch registers receive this byte from the four corresponding bitplanes using the same offset address. For example, if the offset address is 9, then the tenth byte (byte 0 is the first) from the first bitplane will be loaded into the first latch register, the tenth byte from the second bitplane will be loaded into the second latch register and so on for the third and fourth.

The same procedure is used for a write access to video RAM. The contents of the four latch registers are written to the corresponding bitplanes at the specified offset address.



This process involves more than simply reading or writing to the latch register. A read access of video RAM must result in a byte being sent to the processor and a write access must result in a byte being transferred from the processor to the video RAM. However, during a read access, only one byte can be sent to the processor at a time. So, we must determine which of the four bytes in the latch registers is sent. This also applies to writing to video RAM because four different bytes (one for each bitplane) must be transferred from the processor.

The graphics controller's role in graphics programming

The answer to all these questions comes from the nine registers of the graphics controller, which is an important component of every EGA and VGA card. These registers determine how and where all read and write accesses to the video RAM occur. This applies not only to the different graphics modes, but also to the text modes. You may frequently use these registers in graphics programming by assigning them different values, depending on the desired operation. However, this isn't the case

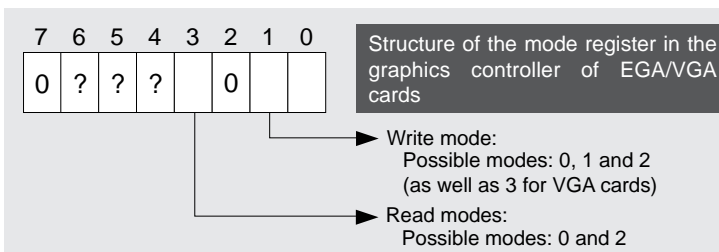
in text mode. In text mode, during initialization of the mode through the BIOS, the various registers are set so additional accesses aren't necessary after initialization. However, all input and output to or from the video RAM in text mode is conducted through the four latch registers, even if this process is transparent for a program. The nine registers of the graphics controller for the EGA card and their defaults:

Reg	Meaning	Default	Reg	Meaning	Default
00H	Set / Reset	00H	05H	Mode	00H
01H	Enable Set / Reset	00H	06H	Miscellaneous	various
02H	Color Compare	00H	07H	Color Don't Care	0FH
03H	Function Select / Data Rotate	00H	08H	Bit Mask	FFH
04H	Read Map Select	00H			

Programming the registers of the graphics controller is similar to accessing the CRTC register of the Hercules graphics card. These registers can also be used on other controllers of an EGA and VGA card. Port address 3CEH has an address register, in which the number of the register within the graphics controller that's being accessed must be loaded first. The value for this register can then be output to the data register next to the address register, which has a port address of 3CFH.

However, access to these two ports doesn't have to be separate. Instead, you can use a 16-bit OUT command on the address register. The AX register sent to this port in the course of the machine language command, OUT DX,AX must contain the register number in the AL register and the value to be loaded for this register in the AH register.

Although values can be loaded in the individual registers of the graphics controller in this way, a read access is only possible with VGA cards; you cannot read these registers on EGA cards.



The mode register, number 5, is important to the graphics controller during read and write accesses to the video RAM. This register sets one of two read modes and one of three write modes. These modes affect read or write accesses to the video RAM. The other registers are only accessories, which specify given parameters depending on the set read and write mode.

In the following sections we'll discuss how the various read and write modes operate, their tasks and how other registers of the graphics controller affect these modes. However, you will quickly realize that you need to use only a few of these modes because the other modes are too difficult to use.

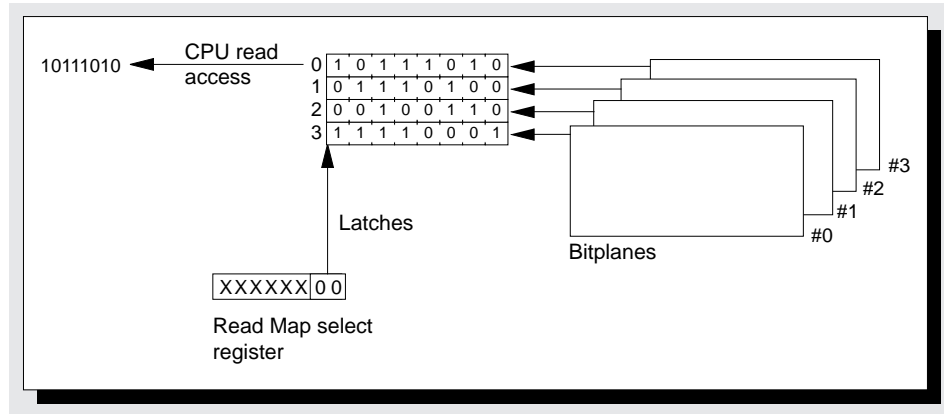
Read mode 0

Read mode 0 gives a program the option of reading a byte from a specific bitplane. This is practical, for example when a part of the video RAM must be saved, for which purpose the four bitplanes are executed sequentially in a loop and the desired area from each bitplane is loaded and placed in main memory.

During a read access to the video RAM in read mode 0 all four latch registers are loaded with the addressed byte of their plane. However, only one of these four bytes gets by its latch register and reaches the CPU. Which bitplane the byte comes from is determined by the contents of the read map select register, which is number 4 within the graphics controller.

Only the lower two bits in this register are allocated. These two bits decide on the number of the latch register, whose contents can advance up to the CPU. When programming this register, ensure the first bitplane is set to 0 (i.e., don't set it to 1 if you want to access the first bitplane).

Read access to the video RAM in read mode 0



The following sequence of assembly commands demonstrates how read mode 0 is used. The first 8K from the second bitplane are loaded into the main memory. First read mode 0 is set and then the read map select register is loaded with the value 1 so the second bitplane can be read. With the help of the assembly command REP MOVSB, the first 8K are then loaded byte by byte from the video RAM and copied to a buffer.

```

mov ax,ds          ;ES:DI to target buffer
mov es,ax
mov di,offset buffer
mov ax,0A000h      ;DS:SI to starting address of bitplane
mov ds,ax          ;in video RAM
xor si,si
mov cx,8*1024      ;Copy 8K

mov dx,3CEh        ;Address graphics controller
mov ax,0005h       ;Write read mode 0 in mode register
out dx,ax          ;
mov ax,0104h       ;Write 1 (plane number in read map
out dx,ax          ;register

rep movsb          ;Copy 8K

```

After studying the assembly sequence, you may think that you could speed up the copying process by simply copying 4096 words instead of 8192 bytes. To do this you would have to convert REP MOVSB to REP MOVSW and load the CX register with 4096 instead of 8192. This would work for copying memory areas within the main memory. However, in this case, it would result in some unpleasant consequences.

Because 16-bit accesses aren't possible in video RAM, the CPU would first execute two read accesses at byte level before executing the write access. The second read access would overwrite the results of the first read access within the latch register before these results could even reach the CPU. As a result, in the target buffer, you would find only words whose low and high byte have the same value. Only the high byte would correspond to the actual contents of the video RAM. For this reason, 16-bit read accesses to video RAM should be avoided in all forms on EGA and VGA cards.

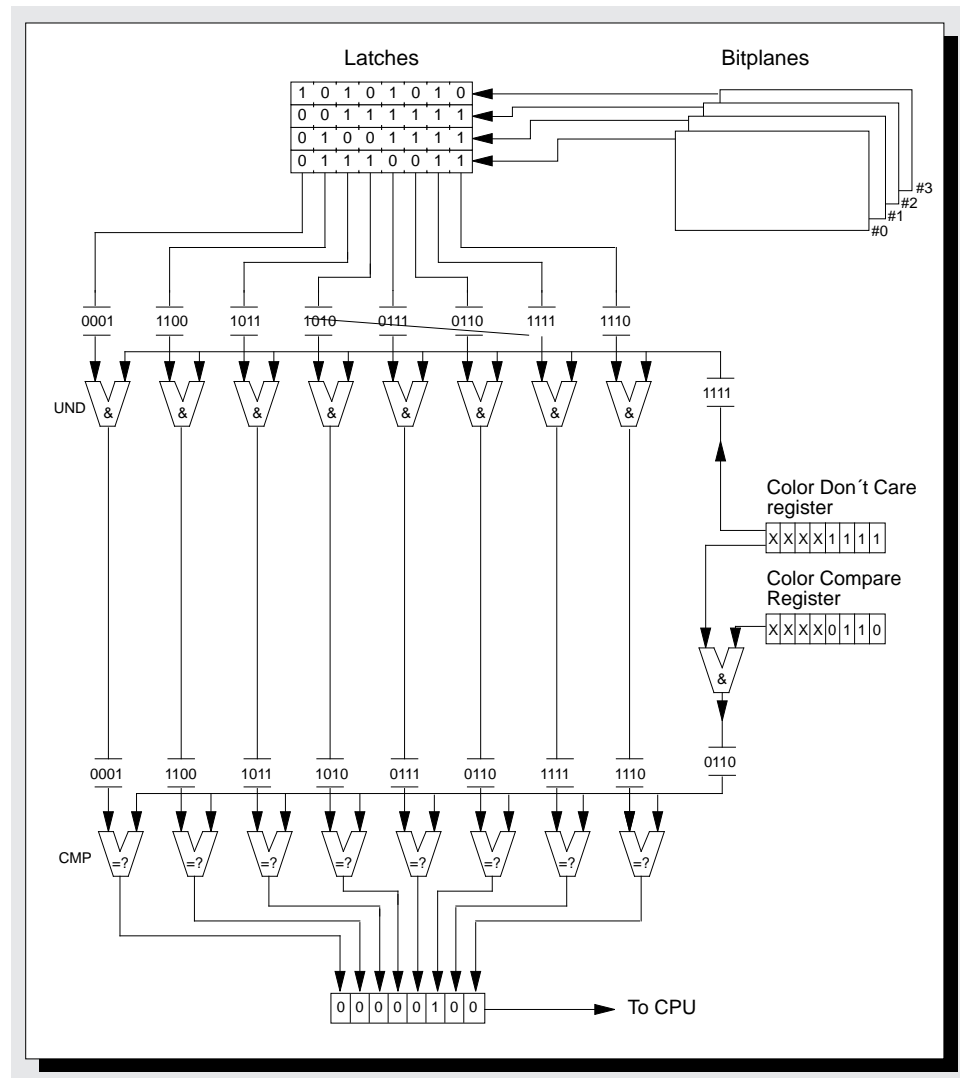
Read mode 1

Although it's relatively easy to understand the operation and purpose of read mode 0, read mode 1 is more complicated. This mode involves more than sending the contents of one of the latch registers to the CPU. Instead, in this mode numerous logical operations, which affect the contents of all four latch registers, occur.

This mode is responsible for determining whether the bits from the four latch registers contain a specific value. Later, this mode can be used in the graphics modes of the EGA and VGA cards with 16 colors to search for pixels with a specific color value. As a rule, however, this mode is rarely used. After the four latch registers are loaded, eight groups, each consisting of four bits, are formed. The four bits occupying an identical bit position within the four different latch registers are located in one group. For example, all the bits at bit position 0 are in one group, all the bits from bit position 1 are in another group, etc.

Each of these eight groups of four bits is now compared with the value from the color compare register, which was loaded into this register before the read access. The result of this comparison determines the byte that is sent to the CPU as the result of the read operation. All the bits, whose group showed the value from the color compare register, are set to 1, while the other bits contain the value 0. So, after the read access, it's possible to determine not only whether one of the groups corresponded to the value from the color compare register, but also which groups correspond.

Read access to the video RAM in read mode 1



Although it may be difficult to believe, this isn't the entire process. Actually, the Color Don't Care register is also involved. Only when this register contains the value `00001111b` are the various groups of four bits completely compared with the

comparison value from the Color Compare register. Each of the lower four bits in the Color Don't Care register represents one of the four bitplanes: Bit 0 for the first, bit 1 for the second, etc. Only if one of these bits contains the value 1 is the corresponding bitplane also included in this comparison of the four bit groups with the value from the Color Compare register.

However, if the value is 0, it's as if the value from this bitplane matches the corresponding bit from the Color Compare register in all eight groups. So, specifying the value 0 in the Color Don't Care Register always returns the value 1111111b to the CPU, regardless of the contents of the four latch registers and Color Compare register. This is possible because the eight groups are no longer compared with the comparison value. Instead, all the groups are considered to be appropriate.

The following assembly sequence demonstrates the use of read mode 1. This assembly sequence establishes which of the groups of four, from the first byte in the four bitplanes in video RAM, has the value 5. The Color Don't Care register isn't explicitly programmed, because it's assumed that its default value is 0000111b. So, all the bitplanes will be included in the comparison.

```

mov ax,0A000h      ;Set ES to video RAM
mov es,ax

mov dx,3CEh        ;Address graphics controller
mov ax,0805h        ;Write read mode 1 in mode register
out dx,ax
mov ax,0502h        ;Write color value 5 in color compare
out dx,ax           ;Register

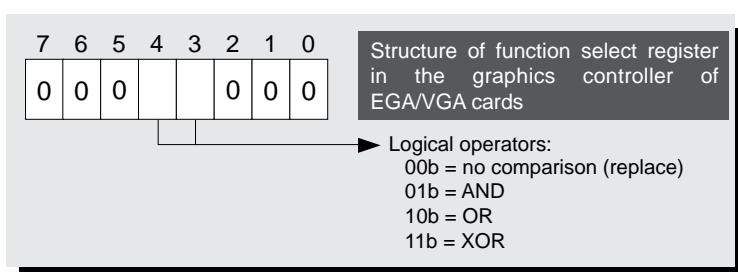
mov al,es:[0]       ;Read and compare pixels,
                    ;return result in AX
or  al,al           ;No bit to 1?
je  AllUnequal      ;No

```

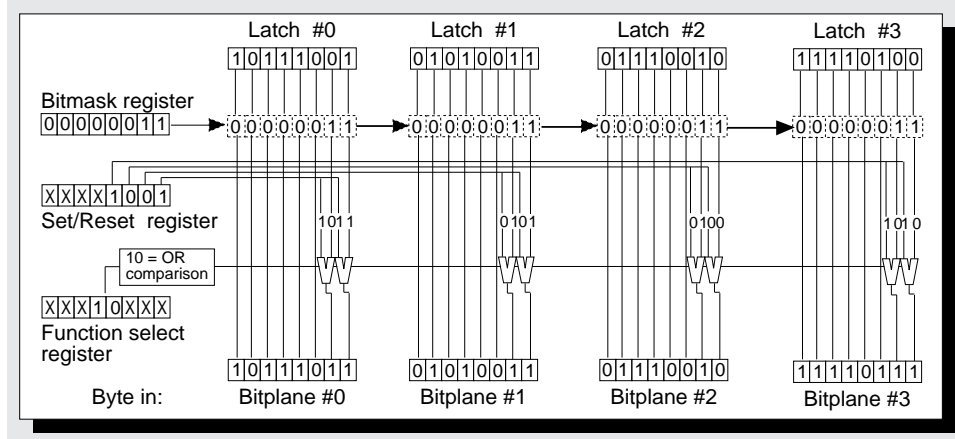
Write mode 0

In accessing the video RAM in write mode 0, several operations, which depend on the contents of several registers, occur. The contents of the bit mask register decide whether the contents of a bit in the four latch registers will go to the four bitplanes unchanged or will be manipulated beforehand. The individual bits in the bit mask register correspond to the bits in the four latch registers. If a bit in the bit mask register contains the value 0, the matching bit in the four latch registers is taken, unchanged, in the four bitplanes. An operation occurs if the bit contains the value 1 instead. Which operation is determined by the contents of the function select register. As the illustration on the left shows, the bits can simply be replaced or manipulated with the help of one of the logical operators AND, OR or EXCLUSIVE OR.

The contents of the Enable Set/Reset register determine what will be the partner of these bits in the operation. If the lower four bits each contain the value 1, the operation takes place through the contents of the lower 4 bits of the Set/Reset register. Each of these bits is used in the operation with the four bits of a bit position from the four latch registers, whose type is described by the contents of the function select register.



Write mode 0
when the
Enable
Set/Reset
register contains
the value
00001111b



All the bits to be manipulated from latch register 0 are then linked with bit 0 of the Set/Reset register by the selected operator. In the same way, all the bits to be manipulated from latch registers 1, 2 and 3 are linked with bits 1, 2 and 3 from the Set/Reset register. The CPU byte, which is transferred to the graphics controller during a write access, is unimportant here. The write access is reduced to a trigger, which cannot have any direct influence on the contents of the latch register (or the bitplanes).

The following assembly language sequence assigns code 1011b to the group of four with bit 2 from the first byte in the video RAM, without disturbing the contents of the other groups. As you'll see in the next section, this is a technique that's frequently used in setting pixels in the 16 color graphics modes of EGA and VGA cards.

Since the color of the other groups of four shouldn't be changed, their contents are first loaded into the latch register through a read access to the video RAM. Which of the various read modes is active is unimportant. After all, the value returned to the processor is not important; it simply fills the latch register.

Since only the group of four at bit position 2 must be manipulated and the other groups return to the bitplanes unchanged, the value 00000100b (04H) is loaded into the bit mask register first. Then, the value 0 is written in the Function Select register because the bits to be manipulated should be replaced by a new bit combination. After that, the color for the group of four at bit position 2 (1011b = 0BH) is loaded into the Set/Reset register.

To remove the color from this register when writing to the video RAM, write the value 1111b (0FH) to the Enable Set/Reset register as the last access to the register of the graphics controller. Then execute the write access to the video RAM, in which the transferred processor byte is unimportant.

```

mov ax,0A000h      ;Video RAM segment address
mov ds,ax          ;to DS
mov al,ds:[0]      ;Load byte 0 in latch register
mov dx,3CEh        ;Address graphics controller
mov ax,0005h       ;Read mode 0, write mode 0
out dx,ax          ;to mode register
mov al,03h         ;Write 0 to function select
out dx,ax          ;register
mov ax,0408h       ;Write bit masks in bit mask register
out dx,ax
mov ax,0B00h       ;Write new color value to
out dx,ax          ;Set/Reset register
mov ax,0F01h       ;Write 1111b to Enable Set/Reset

```

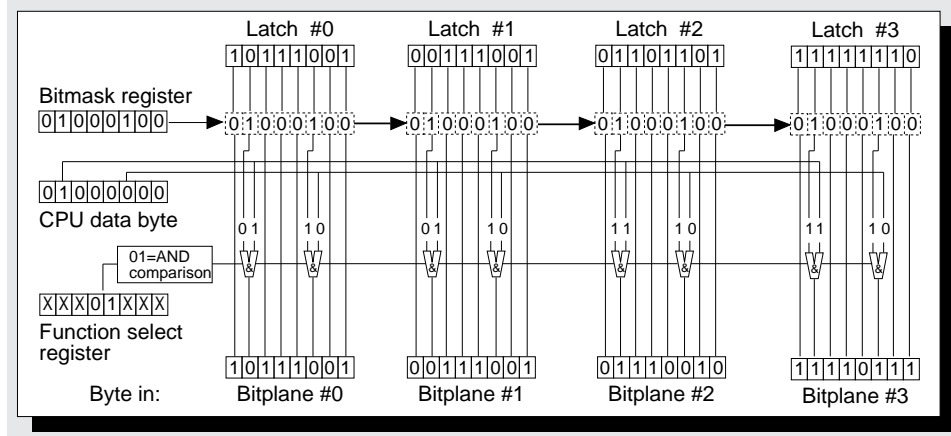


```

out dx,ax          ;register
mov ds:[0],al      ;Manipulate & return latch register

```

*Write mode 0,
when the
Enable
Set/Reset
register contains
the value
00000000b*



This process is different when the Enable Set/Reset register contains the value 0. In this case, all the bits to be manipulated from the four latch registers are linked to the CPU byte latch by latch. Again, the operator depends on the contents of the Function Select register. For example, if you choose OR and want to manipulate bits 1, 2, 4 and 6, then these bits in all four latch registers are individually linked by a logical OR with bits 1, 2, 4 and 6 in the CPU byte.

In this mode the single bit positions in the four latch registers are linked with the same value from the CPU byte. So, this mode isn't used as much as the operation through the Set/Reset register as described.

Write mode 1

Compared to the complex operations in write mode 0, write mode 1 seems very simple. The contents of the various registers of the graphics controller and the passed CPU byte are no longer important here, since the contents of the four latch registers are written to the specified offset address within the four bitplanes in unchanged form.

For example, it makes sense to use this write mode to copy a specific area from the video RAM to another area. Simply run this area in the video RAM, fill the contents of the four latches through a read access in any read mode and then write the latch register to video RAM through a write access in write mode 1. This allows you to copy four bytes at a time, which saves a lot of time.

```

mov ax,0A000h      ;Segment address of video RAM
mov ds,ax          ;to DS and ES
mov es,ax
mov si,0000h       ;Source begins at 0000H
mov di,0200h       ;Target begins at 0200H
mov cx,100h        ;Copy 256 bytes
cld                ;Increment on string inst.

mov dx,3CEh        ;Address graphics controller
mov ax,0105h       ;Read mode 0, Write mode 1.i).Video RAM:Write mode 1 ;
out dx,ax          ;to mode register

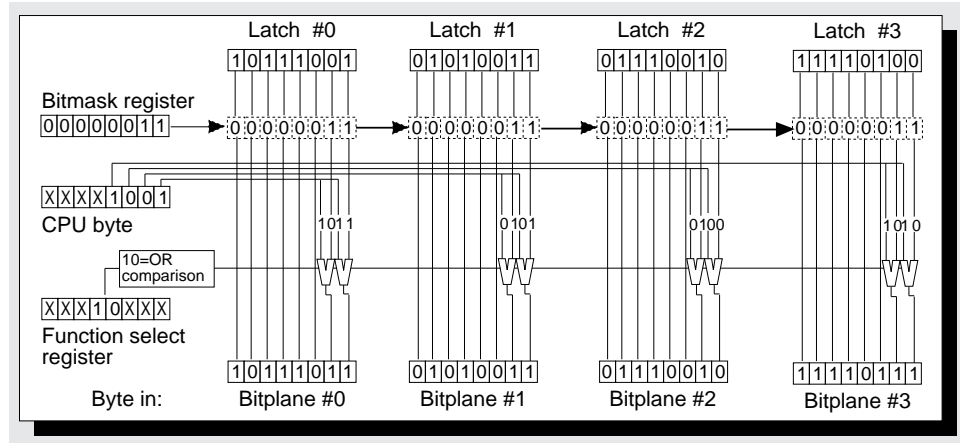
rep movsb          ;Copy all four bitplanes

```

Write mode 2

Write mode 2 is like a combination of the different modes in write mode 0. Like write mode 0, in write mode 2, the bit mask register decides which bits are sent unchanged to the latch registers and which ones are to be manipulated.

Write access to the video RAM in write mode 2



The linking mode recorded in the Function Select register determines the manner in which the single bits are manipulated. Regardless of the contents of the Enable Set/Reset register, the lower 4 bits of the CPU byte are linked with the latch registers within write mode 2. Bit 0 of the CPU byte is linked with all bits in latch register 0 that are to be manipulated. The same applies for CPU bits 1, 2 and 3, which are linked individually with all the bits in latch registers 1, 2 and 3.

This mode is well-suited for setting the color of single pixels in 16 color graphics mode on EGA and VGA cards. While this is also possible in write mode 0, the appropriate assembly sequence is shorter in write mode 2, since neither the Enable Set/Reset register nor the Set/Reset register must be programmed.

Here's the same example for write mode 2:

```

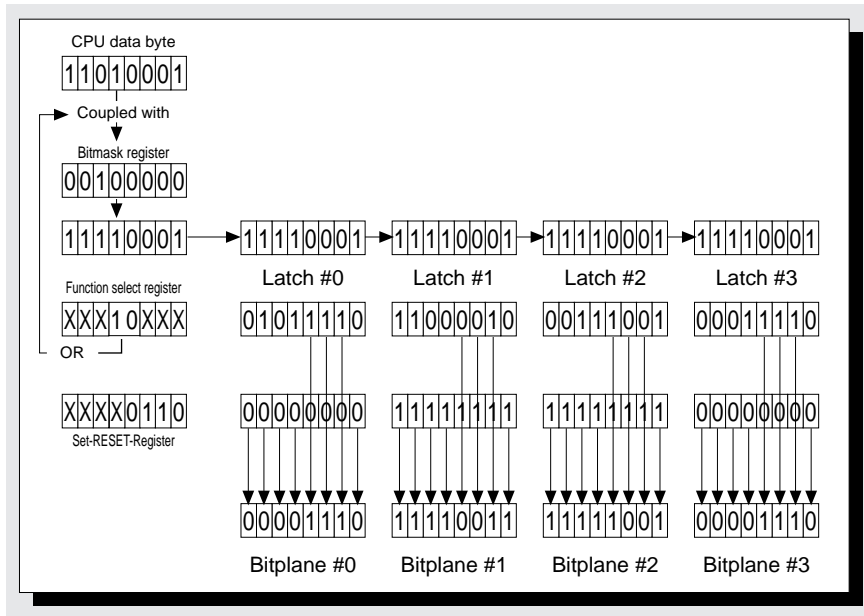
mov ax,0A000h      ;Segment address of video RAM
mov ds,ax          ;to DS
mov al,ds:[0]      ;Load byte 0 to latch register
mov dx,3CEh        ;Address graphics controller
mov ax,0205h       ;Write read mode 0, write mode 2
out dx,ax          ;to mode register
mov ax,0003h       ;Write REPLACE mode (0) to
out dx,ax          ;function select register
mov ax,0408h       ;Write bit mask to bit mask register
out dx,ax
mov byte ptr ds:[0],0Bh ;New color value in video RAM

```

Write mode 3

In addition to the three write modes of the EGA card, the developers of the VGA card added a fourth mode. This mode is selected in the same way as the others; enter the corresponding mode number in the mode register of the graphics controller.

Write access to video RAM in write mode 3



During a read access to video RAM in this mode, the four low bits from the set/reset register must first be coupled with the bits from the four latch registers. Unlike write mode 0, the contents of the enable set/reset register are included in this operation. However, the type of coupling is determined by the Function select register as usual.

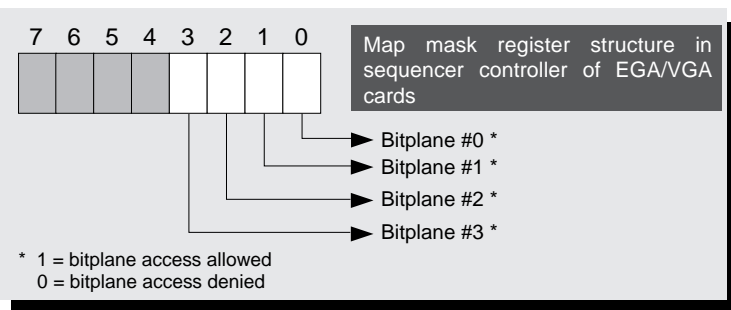
In this mode, a logical AND combines the CPU byte with the contents of the bit mask register. The result then determines which bits from the latch registers are written to the bitplanes unchanged and which are taken as the combination of the set/reset register and the latch bits. The CPU byte is helping in the task the bit mask register performs alone in modes 0 and 2.

Since we have yet to find a meaningful application for this mode, we couldn't provide a practical example.

The map mask register

Another register we'll discuss is the map mask register of the sequencer controller. It's used to lock individual bitplanes which then prevents them from being written to or read. This is useful when you want to manipulate the contents of only one specific bitplane.

Each bitplane's status is represented in this register by the four low bits. Each bit corresponds to a bitplane. The bit must be set to 1 to allow access to the corresponding bitplane.



The following sections describe how to implement the various read and write modes to perform tasks such as setting and querying pixels in graphics modes or copying areas of the screen from video RAM to main memory.

The 16 color EGA and VGA graphics modes

The most important feature of EGA and VGA cards is the 16 color graphics modes. These modes are available with resolutions from 320x200 to 640x350 pixels. For those used to working with the four colors of the CGA card, the 16 color modes are a definite improvement. Although EGA and VGA are still far from the ultimate goal of photographic quality screen displays, which would require approximately 16 million colors, the color graphics modes are a start.

Why multiple 16 color graphics modes?

The development of PC video cards includes a steady progression of higher resolutions with more colors. Why then would the EGA card want to include two lower resolution 16 color graphics modes in addition to the high resolution 640x350 mode? As you'll see in the following table, the lower resolution modes require less memory per screen page. This allows more pages to be stored in video RAM simultaneously. In many user applications, a greater number of screen pages in video RAM may be more important than higher resolution. One example of this is sprite programming, which we'll discuss in "Sprites" later in this section. This section also explains how the remaining video RAM can be used.

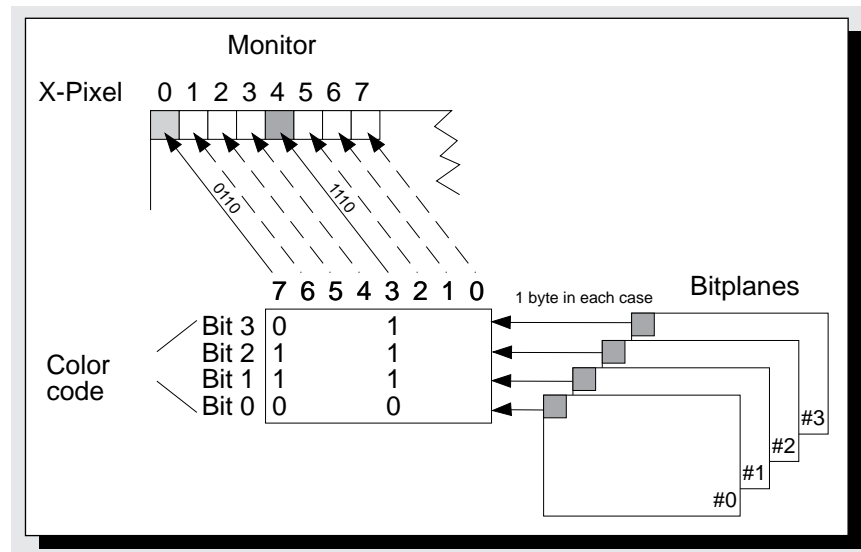
As you can see in the table on the right, the 16 color graphics modes are also fully supported by the VGA card. The VGA card also has an additional 640x480 mode. Although this resolution is exceeded on the Super VGA card, it remains the highest resolution mode for the standard VGA card and has been widely used by many applications.

Video RAM structure in 16 color graphics modes				
Mode	Resolution	Memory	Pages	Remaining bytes
0DH	320x200	32,000	8	6,144
0EH	640x200	64,000	4	6,144
10H	640x350	112,000	2	38,144
12H*	640x480	153,600	1	108,544
*VGA only				

Although we have different modes with different resolutions, the color information for each pixel in all modes is encoded in basically the same way, except for a few minor differences. These differences involve the way video RAM is organized.

Each byte of video RAM contains color information for eight consecutive pixels in the same pixel line of the screen. This means that each pixel is represented by one bit. In the 16 color modes this isn't sufficient because each pixel needs four bits to identify the color. These bits are obtained by using one bit from each of the four bitplanes, as shown in the following illustration.

Structure of video RAM in EGA and VGA 16 color graphics modes



Let's look at a practical example. The first eight pixels in the upper-left corner of the screen are represented by the four bytes found at offset address 0000H in each of the four bitplanes. The four bits required to encode the color information are obtained by combining the bits at the same bit position in the first byte of each bitplane. The procedure creates eight groups of four bits, each of which determines the color of a single pixel.

The bit from bitplane 0 becomes bit 0 of the color code. The bit from bitplane 1 is bit 1 of the color code and so on for bits 2 and 3. The color code is then interpreted as a color, just as with the CGA card: black (0), blue (1), green (2), etc.

We've discussed how the division of video RAM into bitplanes works in the various read and write modes. The same groups of four bits are at work with the read and write modes, accessing individual pixels using the latch registers in 16 color graphics modes.

In addition to the structure of each byte, the order in which these bytes are stored in video RAM is an important aspect of programming the graphics modes. EGA and VGA cards use a very simple procedure for this in the 16 color modes. Starting at offset address 0000H (the start of video RAM), each pixel line is represented by a certain number of consecutive bytes. For graphics modes with a horizontal resolution of 640 pixels, this is 80 bytes. Mode 0DH (320x200) uses 40 bytes per pixel line.

Pixel lines follow one another sequentially in memory, so you can use the following formula to calculate the offset address of the byte that contains any given pixel:

$$\text{Offset} = Y * (\text{horizontal_resolution} / 8) + \text{int}(X / 8)$$

The byte found at this address contains information for eight consecutive pixels. To find the specific bit that contains the color information for the desired pixel, use the following formula:

$$\text{Bit} = 7 - (X \bmod 8)$$

The difference between the various 16 color graphics modes lies in the length of a graphic line in RAM, the number of lines per screen and the amount of memory required per screen page instead of in how the color information is encoded. This information must be considered when programming each of the various graphics modes.

Accessing individual pixels in 16 color graphics modes

The V16COL.PAS, V16COLPA.ASM, V16COLC.C and C16COLCA.ASM demonstration programs which you'll find on the companion CD-ROM show how these formulas are used to access individual pixels through read and write modes. The programs, named V16COLP.PAS and V16COLC.C, can be used with of the EGA/VGA 16 color graphics modes. As we'll see in the "Super VGA Cards" section of this chapter, this also applies to the Super VGA card's 16 color 800x600 mode.

Assembler routines improve both programs' performance. These routines, called V16COLPA.ASM and V16COLCA.ASM, access individual pixels and work with different screen pages.

The high level language and assembler modules use the same global variable and routine names. Both versions of the program work almost identically, so our discussion will apply to both versions. Let's begin with the assembler module, since this is where the work actually occurs. The PUBLIC declarations at the beginning of the module lists the routines used to support the C or Pascal program. You'll find four routines called INIT640480, INIT640350, INIT640200 and INIT320200. One of the four EGA/VGA 16 color graphics modes can be selected using these routines. In addition to these, the routines GETPIX routine; and SETPIX routine; set or read individual pixels in the selected graphics mode. The last two routines in the assembler module are SETPAGE; and SHOWPAGE. These routines are responsible for setting the screen page that is accessed by GETPIX and SETPIX and selecting the page that is displayed.

The assembler module for the C program has another routine called GETFONTPTR. This routine actually has nothing to do with graphics programming. It simply returns a pointer to the 8x8 font stored in the EGA/VGA ROM chip. This pointer is required by the C or Pascal module to display characters on screen in graphics mode. The Pascal version of the program gets this pointer without using an assembler routine, so GETFONTPTR isn't used in V16COLPA.ASM.

The INIT routines for initializing the graphics mode perform two basic tasks. First, function 00H of the BIOS video interrupt sets the desired video mode and clears the screen. Then three global variables are declared for use in accessing individual pixels and screen pages.

The three variables are VIO_SEG, LNWIDTH and PAGEOFS. The first variable, VIO_SEG, stores the segment address of video RAM. This variable also represents the current screen page because the offset of the current screen page is included in this segment address. This means that SETPIX and GETPIX don't have to consider the starting address of the current screen page when addressing a pixel; the segment address in VIO_SEG already reflects this information. For example, in 640x200 pixel graphics mode, each screen page requires 64,000 bytes, which is spread across the four bitplanes. This means that a new screen page starts every 8000 bytes within a bitplane. In hexadecimal mode, this interval is 1F40H bytes.

The first screen page begins at A000:0000, the second at A000:1F40, the third at A000:3E80, the fourth at A000:5DC0, etc. Using the segment address, the first page starts at A000:0000, the second at A1F4:0000, the third at A3E8:0000 and the fourth at A5DC:0000. You must consider the screen size before calculating the start of each screen page for each video mode. The INIT routines store this information in the global variable PAGEOFS, which represents the offset address divided by 16. This division allows easy merging with the segment address.

The global variable LNWIDTH calculates individual pixel addresses. It stores the width of a single pixel line in bytes. This value is plugged into the "horizontal_resolution / 8" expression in the equation previously listed.

With the information stored in the global variables VIO_SEG and LNWIDTH, routines such as SETPIX and SHOWPAGE can perform their tasks. With SETPIX and GETPIX, the screen coordinates specified by the caller are converted into an offset address for accessing video RAM. This is done by multiplying the Y-coordinate by LNWIDTH, dividing the X-coordinate by eight and adding the two results.

Then the bit position of the desired pixel is calculated from the X-coordinate. In addition to the MOD operation in the equation already listed, the assembler instruction AND is used to ignore all but the three lowest bits of the X-coordinate. This performs the same task as a MOD operation with 8, but runs faster.

Until now, GETPIX and SETPIX are identical in their execution. With SETPIX, the next step is to convert the bit position into a bit mask, which is needed later to access the pixel using write mode 2. This is done by shifting the value 1 to the left by the number of the bit position. For example, by using bit position 4 you would end up with the value 0010000(b). This value is then loaded into both the bit mask register of the graphics controller and the mode register for write mode 2 and read mode 0. Next, the segment address of the video RAM, including the offset address of the current screen page, is loaded in the ES register so video RAM is addressable.

Although we're going to set the pixel without reading it, the next step is to load the byte that contains the pixel to be accessed. The value passed to the CPU is actually meaningless. The purpose of this operation is only to load the four latch registers with the four bitplane bytes that contain the color code for the pixel we want to access. The pixel color is then set by writing the desired color code to video RAM.

According to write mode 2, the EGA or VGA card then writes this bit value in all the latch register groups of four that have a value of 1 in the corresponding bit position of the bit mask register. In our case, this applies only to the bit position of the pixel we're accessing, so all other groups of four remain unaffected by the write access. In the four bitplane bytes, only the four bits that determine the pixel we're accessing are changed in the write operation that automatically follows.

This completes the task of SETPIX. Before returning to the caller, however, the changed graphics controller registers are set back to their default values. This should always be done whenever these registers are manipulated so the next routine that uses them can assume they contain the default values. This allows you to change only the appropriate registers; the others can be left with their default values.

The same applies to GETPIX. However, here only one register of the graphics controller must be programmed. This is the read map register. At the end of the routine, this register is set back to its default value. It's more difficult for GETPIX to read a pixel than for SETPIX to set one. Unfortunately, there is no corresponding read mode in the EGA and VGA cards, so GETPIX must manage with read mode 0.

This mode returns only one byte per read access of the video RAM. This byte comes from the bitplane indicated by the number in the read map register of the graphics controller. So, GETPIX must run through a loop four times, reading a byte from one of the four bitplanes each time. Before this bit can be read, GETPIX programs the read map register within the loop.

Access proceeds from bitplane 3 to bitplane 2, bitplane 1 and finally bitplane 0. At the end of the loop, the read map register is loaded again with its default value of 0. Within the loop, the color of the desired pixel is represented using the four bytes read. All the bits in each byte that don't apply to the desired pixel are hidden using a previously defined bit mask. The result is a byte with either only one bit set or with none. The latter case indicates the corresponding bit from the color code of the pixel wasn't set in the bitplane currently being processed.

The status of this bit is obtained by executing the NEG instruction when other bits have been blanked out with the bit mask. The result of this instruction is the status of the bit not affected by the bit mask is reflected in bit 7 of the register. If the isolated color bit was set, then bit 7 of this register is set after the NEG operation. Similarly, bit 7 is off if the corresponding color bit isn't set.

The bit 7 obtained in this way is moved from the BH register to the BL register with the ROL instruction. This operation is executed four times. The previous result is moved one place to the left to a lower position with each execution of the loop, which then loads the color of the desired pixel in the lower four bits of the BL register. This is why it's important to process the bitplanes from number 3 to number 0; otherwise, if you switched this, you would invert the color code.

This is how GETPIX returns the actual color code of the selected pixel. This method works, even if it's slower and more complicated than the procedure for setting a pixel. The SETPAGE and GETPAGE routines are much simpler than SETPIX and GETPIX. This is especially true for SETPAGE, which sets the current page number in video RAM. SETPIX and GETPIX operate within this routine. The SETPAGE routine works by multiplying the screen page number by the page length as stored in the PAGEOFS variable. The base segment address of A000H is added and the result is stored in the global variable VIO_SEG.

All subsequent calls of SETPIX and GETPIX are then based on this screen page. Remember, a screen page isn't displayed on screen simply by selecting it with SETPAGE. This enables you to select a page in the background and work on it with SETPIX and GETPIX while a different page is displayed.

SHOWPAGE actually displays a page. SHOWPAGE accepts a page number as input and begins by converting this page number into an offset address in video RAM. This operation also multiplies the value of PAGEOFS by the page number. This time, however, the result is also multiplied by 16, because the value in PAGEOFS is a segment address and not an offset address as is required here.

To display a screen page on screen, the offset address of the screen origin is loaded into two registers of the CRT controller. Theoretically, you can load any value you want in these registers. However, for the display to be understandable, the screen origin and the starting address(es) used by routines, such as SETPIX and GETPIX, must be specified. So, the page number is also multiplied by PAGEOFS here.

From the assembler modules V16COLPA.ASM and V16COLCA.ASM we move on to the main program modules, from which the routines previously described are called. Both programs can work in all EGA and VGA 16 color graphics modes, if the mode is set before the program is compiled.

We find a constant called MODUS in the declaration of constants. It must be assigned a value of A320200, A640200, A640350 or A640480. These values indicate the various graphics modes. At runtime, the main program uses the IsEgaVga function to determine if the mode indicated by the value of VMODE can be initialized. For the 640x480 mode, this means that a VGA card must be installed. All other modes will run with either VGA or EGA.

If the installed video card passes this test, the global variables MAXX, MAXY and PAGES are loaded with values that depend on the selected video mode. These variables indicate the maximum X-coordinates and Y-coordinates on the screen and the total number of screen pages available. This information is very important for the DEMO routine, which is called next. DEMO uses the various assembler module routines for running through each screen page in a loop, defining each page with SETPAGE, using each page in conjunction with SETPIX and GETPIX and finally displaying each page with SHOWPAGE.

Finally (also within the loop), the COLORBOX, DRAWAXIS and GRFXPRINT or GRFXPRINTF (in the C version) routines are called to fill the screen.

COLORBOX essentially draws a box filled with lines drawn by the LINE routine. LINE is based on the Bresenham algorithm that draws lines without complicated floating point mathematics. Each pixel in the line is drawn with SETPIX.

Because of the way it works, this routine isn't very fast. However, it is intentionally written in the high level language instead of assembly language. Of course, you can rewrite this routine in assembly language. To do so, you need routines for filling areas, drawing circles and polygons, etc. You'll quickly realize that this requires a lot of work. We didn't think the added performance was worth the extra programming effort, so the C and Pascal LINE routines are suitable for demonstration purposes. In addition to LINE, the GRFXPRINT and GRFXPRINTF functions also use SETPIX. These routines print letters and numbers in graphics modes. Each character is read pixel by pixel from the 8x8 ROM resident font and transferred to video RAM using SETPIX.

You'll find the following program(s) on the companion CD-ROM



V16COLP.PAS (Pascal listing)
V16COLPA.ASM (Assembler listing)
V16COLC.C (C listing)
V16COLCA.ASM (Assembler listing)

Although it's not extremely fast, this program demonstrates how assembler routines can be used in the EGA/VGA 16 color graphics modes. The routines in this demo program serve as a starting point for your own 16 color graphics programs.

The VGA 256 color graphics modes

One of the biggest advantages of the VGA card over its predecessor, the EGA card, is its ability to display 256 different colors on the screen simultaneously. This group of 256 is selected from a palette of 262,000 colors. This was a milestone in the development of PC graphics.

But the 256 color mode has a resolution of only 320x200 pixels, which is much less than the 640x480 high resolution 16 color mode. However, display quality is based on more than resolution. For example, it can also be based on the wide variety of colors in this mode. This section discusses how to select this graphics mode, address individual pixels and increase the actual resolution.

Setting the 256 color mode and addressing pixels

To programmers, the 256 color mode with a resolution of 320x200 pixels is the easiest VGA graphics mode to use. This is especially evident in the way the pixels are addressed. Pixel addressing is similar to text mode access in some ways.

Before we can access any pixels, we must enable this mode. You can use function 00H of the BIOS video interrupt to do this. The number for this mode (13H) is passed in the AL register and function number 00H is passed in the AH register. These are the only requirements for setting 256 color graphics mode. This single operation also sets the graphics controller's read and write modes and the corresponding registers. This prepares the video card for access to the 320x200 pixels.

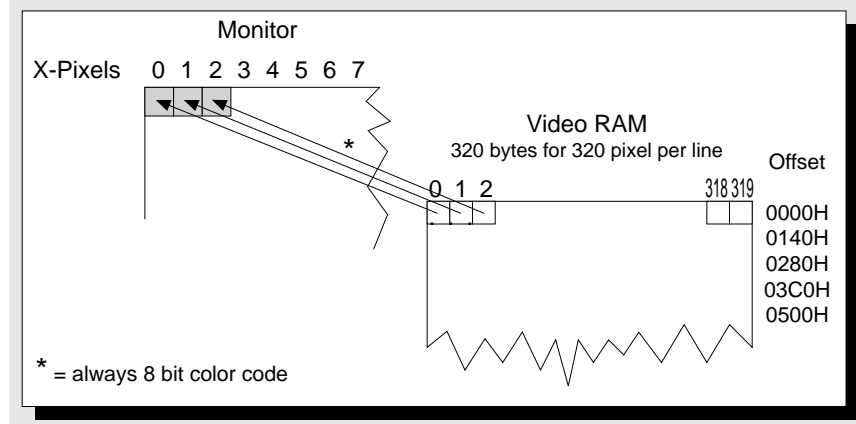
The four bitplanes that were so bothersome in the 16 color graphics modes aren't used; 256 color mode uses a much simpler video RAM organization. Each pixel is represented in video RAM by a single byte, just as in text mode. This byte represents the color as a value between 0 and 255. This color value acts as a direct index to the DAC color table, from which the actual color representing the pixel on screen is taken.

Since a single byte describes each pixel, a screen line consists of 320 contiguous bytes in video RAM. These bytes represent pixels in a row, from left to right. As in text mode, each succeeding line (row) begins immediately after the end of the previous line in the video RAM. So the offset address of a pixel is calculated from its screen coordinates as follows:

$$\text{Offset} = y * 320 + x$$

All pixels are therefore located within the 64K segment of video RAM that begins at segment address A000H in this mode. This makes it easy to access every pixel.

*Video RAM
structure
320x200 pixel
256 color
graphics mode*



Since it's easy to access any pixel in this mode, a special demo program isn't required. However, there's another reason for not including a demo program. Since each pixel is represented by one byte, this only requires 320x200 bytes or 64K. In a fully populated VGA card, this leaves 192K of unused video RAM.

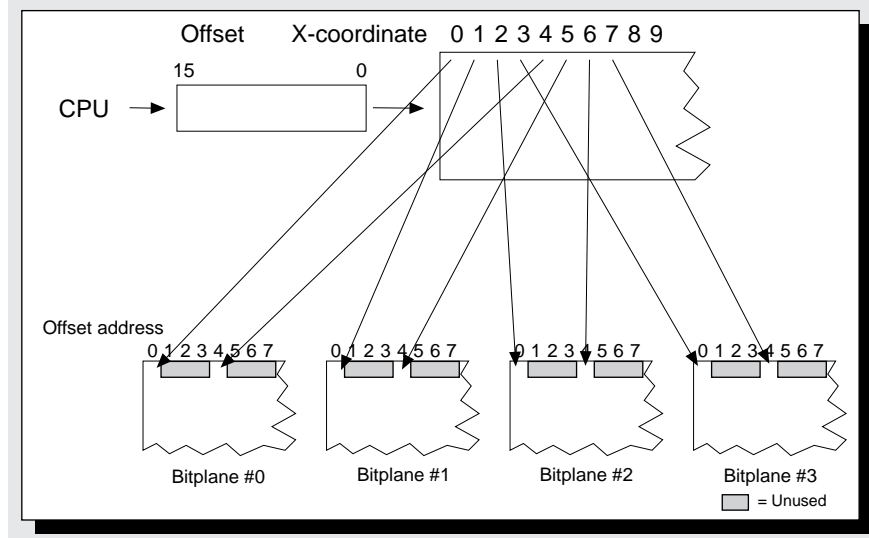
So, why can't we simply include three additional screen pages in the remaining video RAM? Although this is theoretically possible, the video RAM structure previously described won't allow it. This may have something to do with the MCGA card, which is related to the VGA card. The MCGA card has the same 256 color mode the VGA card uses, but it's limited to 64K of video RAM (i.e., one screen page). Perhaps IBM wanted to maintain compatibility between the two cards in this mode.

Four graphic pages in one

To overcome this limitation, you can use a few tricks to store and access four graphics pages in video RAM while in VGA 256 color graphics mode.

To manage four graphics pages in the 256 color mode, you must understand how pixel information can be divided among bitplanes. As in text mode, the contiguous video RAM model beginning at A000H is only an illusion that's created to make it easier for the programmer to address individual pixels.

*Video
information
storage
256 color mode,
320x200 pixel
resolution*



The pixels are actually managed in four bitplanes, just as we saw earlier, using the Chain4 mode. This mode is an extension of the odd/even mode the VGA card uses in text mode to move information from video RAM to bitplanes 0 and 1. In this mode, the lower two bits of a specific offset address determine the number of the bitplane to which the value is sent. These two bits are then internally set to 0 and are used as the offset address to access the selected bitplane. Three bytes remain unused between each occupied byte in each bitplane. Instead of distributing this information among all four bitplanes, like the 16 color modes, a single byte within the bitplane contains the color information for a pixel. So, four pixels can be found at the same offset address, but in different bitplanes.

Let's consider the first line on the screen. The information for the first four pixels in this line is set at offset address 000H in each of the four respective bitplanes. The pixel with X-coordinate 0 is in bitplane 0, the pixel with X-coordinate 1 is in bitplane 1, etc. According to this scheme, we also find the pixels with X-coordinates 4 through 7 at the same offset address, which is the fourth byte in each bitplane. The information for the pixels is spread over the four bitplanes in the same way.

Since the 64,000 bytes needed to store 320*200 pixels are spread out across the four bitplanes, only 16K of each bitplane is actually used. However, there still isn't room for additional screen pages in video RAM. This occurs because the bytes are stored across the entire bitplane with permanent "gaps" of three bytes between each byte that's actually used.

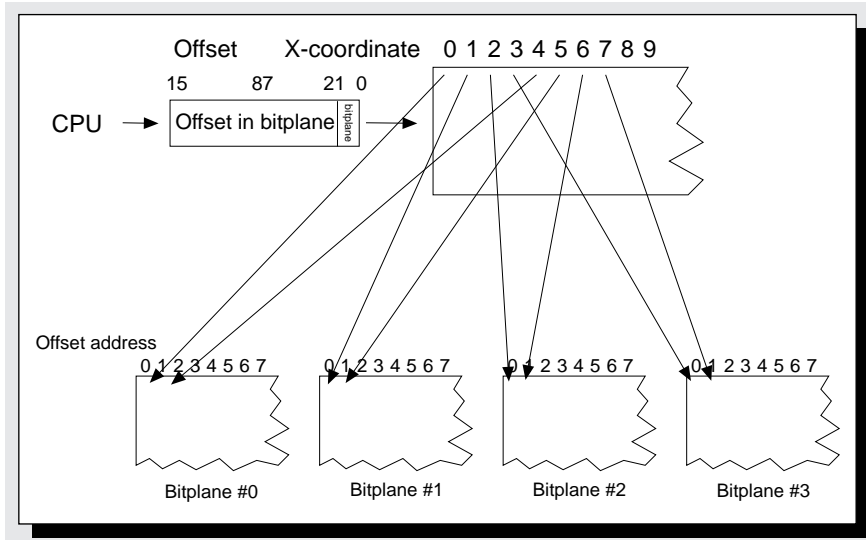
To store more than one page in the video RAM at a time, we must have a way to move the occupied bytes closer together so we don't waste three bytes for every byte used. To do this, you must reprogram several VGA registers. However, to do this, we must omit Chain4 mode. This means that we must write the color information to the various bitplanes "by hand". The demonstration programs V3220C.C, V3220P.PAS, V3220CA.ASM and V3220PA.ASM control VGA 320x200 pixel graphics mode. The assembler modules contain various routines for initializing video mode and accessing individual pixels. They are specially developed to be called from the accompanying high level language module.

Both programs use the init320200 assembler routine to configure the 320x200 256 color graphics mode that allows four screen pages in video RAM. This operation begins by setting the video mode as usual with BIOS function 13H. Then the routine changes the registers that are needed to restructure the video RAM according to our needs.

The first step is to switch off the Chain4 and the odd/even modes. This activates a sort of linear mode, which means the offset addresses aren't grouped into bitplanes when video RAM is accessed. This affects read and write access to video RAM only from the CPU's point of view. To the CRT controller, nothing has changed yet.

The CRT controller must be informed the color bytes in the bitplanes are adjacent, rather than spaced in four byte intervals. Next, we switch from doubleword to byte mode, which means that word mode must also be disabled. The program listings document the registers involved with each change. Refer to the end of this chapter for detailed descriptions of the EGA and VGA registers. After completing this operation, the CRT controller views video RAM as follows:

Altered 320x200 pixel mode as seen by the CRT controller



Now, since only the first 16K in each bitplane are occupied, it's possible to store three additional screen pages in the video RAM. The first screen page begins at offset address 0000H in each bitplane, the second at 4000H, the third at 8000H and the fourth at C000H.

In this mode, it's more difficult to address each pixel because all the pixels aren't available in the 64K of video RAM at A000H. Each bitplane is addressed individually. The `setpix` routine; in the assembler module performs this task. It expects the X- and Y-coordinates and the pixel color as its arguments.

`Setpix` starts by creating the offset required to access the desired pixel. According to the new video RAM structure, each bitplane represents 80 pixels instead of 320. So, we begin by multiplying the Y-coordinate by 80. Then, the X-coordinate is divided by four because there are still four consecutive pixels located at each offset address. The sum of these two calculations can then be used to access the proper bitplane.

The number of the bitplane to be accessed is taken from the two lowest bits of the X-coordinate. These two bits represent the number of bits the value 1 is moved to the left to create the bit mask for the map mask register. After this bit mask is passed to the proper register, all subsequent write accesses are addressed to the correct bitplane. The offset address previously calculated is then used to access the desired pixel and assign the given color to it. Now, all we have to do is determine how the different screen pages are selected. Instead of being selected by `setpix`, the screen page is selected by a previous call to `setpage`. The `..setpage` routine; determines the screen page, to which all subsequent calls of `setpix` will apply. Another `setpage` call is required to change the page.

The current page is passed to `setpix` using the segment address stored in the `vio_seg` variable. This variable stores the video RAM segment address of the page selected by `setpage`. This is A000H for the first page, A400H for the second, A800H for the third and AC00H for the fourth. Since the segment address already contains the screen origin, it doesn't have to be considered again in calculating the offset address in `setpix`.

To determine the color of a given pixel, the `getpix` routine is included in the assembler module. The arguments for this routine are the X- and Y-coordinates for the desired pixel. The pixel color is returned as the result.

The `getpix` routine works similar to `setpix`, except that we don't have to program the map mask register of the sequencer to determine the bitplane to be accessed. Instead, we're interested in the read map register of the graphics controller. This register determines the contents of which latch register and therefore which bitplane, is read and sent to the CPU during a read access to video RAM in read mode 1. Unlike the map mask register, the input for this register is only the value of the desired bitplane, instead of a bit mask.

After the offset address of the desired pixel has been calculated and the read map register has been programmed, the pixel color can be read from video RAM. Again, the segment address is taken from the contents of the `vio_seg` variable so the screen page determined by the last `setpage` call is used.

One more routine is needed to display the selected page on screen. This is the `showpage` routine. Its argument is the number of the page to be displayed. `Showpage` then sends the desired page to the screen by loading the offset address 0000H, 4000H, 8000H or C000H in the starting register of the CRT controller.

The demo program shows what you can do with the routines in the assembler module. Each of the four screen pages is loaded with a very similar pixel pattern, consisting of a coordinate grid, a copyright message and an object drawn with various lines. These lines use a series of colors with color numbers from 0 to n. The `ColorBox` routine draws this object. The variable n represents the upper limit of the color number. This is set to 16 on page 0, 64 on page 1, 128 on page 2 and 256 on page 3. `ColorBox` uses a routine called `Line` to draw the various lines. This routine accesses `setpix` from the assembler module to draw lines according to the Bresenham algorithm.

The four different screens are identical except for minor details. The main program quickly switches between them to create interesting optical effects.

You'll find the following program(s) on the companion CD-ROM



V3220P.PAS (Pascal listing)
V3220PA.ASM (Assembler listing)
V3220CC.C (C listing)
V3220CA.ASM (Assembler listing)

320x400 pixels with two screen pages

Although the 256 colors of the 320x200 mode are impressive, the resolution of this mode doesn't compare well with that of other VGA graphics modes. VGA Mode 12H, the highest of the standard VGA modes, uses almost five times as many pixels on screen. Although the many colors of the 256 color mode give the impression of higher resolution, as with a television, it still would be nice to have more pixels with which to work.

When you consider that a single screen page requires only 64K of video RAM in this mode, you may start to wonder if it isn't possible to use more memory per screen page and increase the number of pixels per page. As we'll see later, the VGA card can be programmed to display 400 pixel lines per screen instead of 200. Actually, this is easy to do because the 320x200 mode doesn't actually use 200 horizontal pixel lines.

Even in the 320x200 pixel mode, the VGA card actually has 400 horizontal lines on screen. Since the lines are joined in pairs, it appears as if only 200 lines are on the screen. So, instead of having to double the number of lines, we must simply address each pixel line separately.

The CRT registers responsible for the horizontal and vertical timing don't have to be reprogrammed to change the 320x200 mode to 320x400. Although this results in a rather unusual ratio between the horizontal and vertical axes, the doubled resolution makes this a problem we can tolerate.

The 128,000 pixels in a single screen cannot all be addressed in the usual way. This would require 128K of video RAM, which is too much for the Chain4 mode. So, we use the same method described in the last section to address the pixels in this mode. Each screen page requires 128K, so there is enough room for two pages in video RAM. For most applications, this is sufficient.

Two demonstration programs on the companion CD-ROM (V3240C.C and V3240P.PAS) show you how graphics programming works in this mode. These programs are very similar to the ones discussed in the previous subsection. Each program also uses an assembler module (V3240CA.ASM and V3240PA.ASM). Compared to those already presented, these modules needed only slight modifications. These changes reflect the differences between the 320x200 and 320x400 modes.

You'll find the following program(s) on the companion CD-ROM



V3240P.PAS (Pascal listing)
V3240PA.ASM (Assembler listing)
V3240C.C (C listing)
V3240CA.ASM (Assembler listing)

One difference is immediately noticeable. The init320200 routine has been replaced with a routine called init320400. The routine itself has changed very little. Video mode 13H is still initialized using the BIOS. The linear mode for addressing video RAM is selected and doubleword mode is replaced with byte mode.

A new step in the routine is the access to the maximum scan line register of the CRT controller. Two changes occur: the bits that indicate the number of pixel lines and the character height. The 200 line bit is disabled so all 400 lines are displayed. The other bit must be set from 1 to 0 to indicate that each pixel line will be processed individually instead of in groups of two. These are the only changes required to switch from 320x200 mode to 320x400 mode.

The change in screen resolution also affects the setpage and showpage assembler routines. These routines are changed to reflect the new screen page size. Also, the second (and last) screen page now begins at 8000H instead of 4000H. Surprisingly, setpix and getpix don't have to be changed at all. The line length remains the same and the lines are still stored contiguously in video RAM. Now there are twice as many lines as before and they can be accessed by the correspondingly greater Y-coordinates.

The main programs are also similar. A coordinate grid is drawn on screen along with a copyright message. A box filled with colored lines appears below this. You can immediately tell there are 400 pixel lines on screen by looking at the coordinate grid. You can also tell that more than one screen page is in memory because of the interesting effects that result from quickly switching the two pages.

Finally, we should mention that some (not all) VGA cards have another 256 color mode, with a resolution of 360x480 pixels. This mode exceeds the limit of 128K per screen page, so only one page can be stored in video RAM at a time. For these reasons we won't discuss this mode in detail. The practical resolution limit of the VGA card with 256 colors lies with the 320x400

mode. If you really need greater resolution in a 256 color mode, you should consider using the Super VGA card, which offers higher resolution modes as a standard feature.

Freely selectable colors

An important difference between the EGA/VGA cards and their predecessors is the ability to work with more than just the 16 basic colors. The EGA card has 64 colors from which to choose and the color palette of the VGA card exceeds 262,000 different colors. Of course, not every available color can be displayed on screen at one time. Depending on which text or graphics mode you're using, you'll be limited to a set of 16 or 256 colors that can be used at any given time.

What are palette registers?

The 16 palette registers are part of the attribute controller. These registers are important for the color display in all text and graphics modes with 16 or fewer colors. When the CRT controller is building a screen, it receives the color information for a given pixel as a value between 0 and 15. This value is used as an index into the palette register table. The color information is taken from the index palette register and sent directly to the monitor by the EGA card.

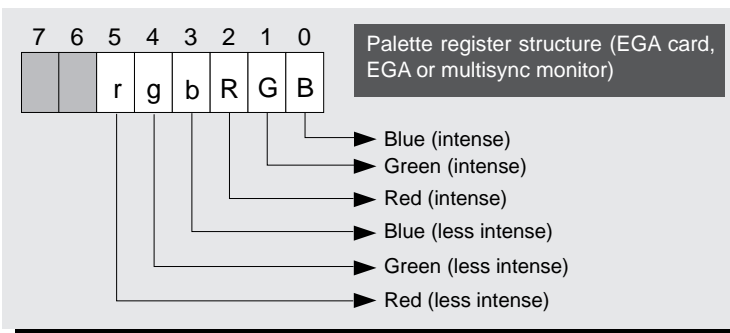
This process shows how the EGA and VGA cards still have a strong connection between the different color codes and the colors that appear on the screen, similar to the earlier video cards. So the programmer has the option of freely selecting the colors that should appear on the screen. This global color selection must occur without changing the contents of video RAM. For example, if all pixels in 640x350 pixel graphics mode were black and a program abruptly changed these pixels to another color, changing the palette register contents from 0 to another value is all that's needed.

If you allow for the palette registers, VGA and EGA color selection isn't different than CGA color selection. The color values are identical to CGA after initializing the palette registers by calling BIOS video interrupt function 00H. The palette registers can then be controlled using the sub-functions of BIOS video interrupt 10H, function 12H, after video mode initialization using function 00H.

To call this function, place the function number 12H in the AH register and the sub-function number (31H) in the BL register. The AL register, which normally contains the sub-function number, is used here to determine whether the palette registers are automatically initialized. If this register contains a value of 1, the palette registers aren't initialized with each subsequent call of function 00H. A value of 0 switches on the automatic initialization feature.

With this feature switched on, you can use the expanded color capabilities of the EGA and VGA cards by programming the palette registers. Before doing this, you must understand the structure of the palette registers. With an EGA card and an EGA or multisync monitor, the individual bits in a palette register directly correspond to the different monitor leads that encode the colors. The basic colors red, green and blue (RGB) each have two leads available. One represents a brighter, more intense display and the other is for a normal display. This makes a total of six bits involved in color programming, which allows for a maximum of 64 (2⁶) colors; 16 of these colors can be loaded in the 16 palette registers at a time.

The attribute controller's color plane enable register plays a vital role in color selection and the palette registers. Before every access to one of the 16 palette registers, the EGA/VGA video controller executes a logical AND between the color index and the lower four bits of this register.



This operation is usually transparent because the lower four bits of the color plane enable register contain a default value of 1111(b). The AND operation with the color index doesn't change the value and the desired color appears on screen.

This is quite different if the value in the color plane enable register changes. For example, if a value of 0111(b) is stored in the color plane enable register, the AND

operation with the color value would result in the highest bit from the color value being switched off. This means that all pixels (or characters) with color codes from 8 to 15 would be reassigned to color codes from 0 to 7. Actually, this capability is seldom used and the default value of the color plane enable register is rarely changed.

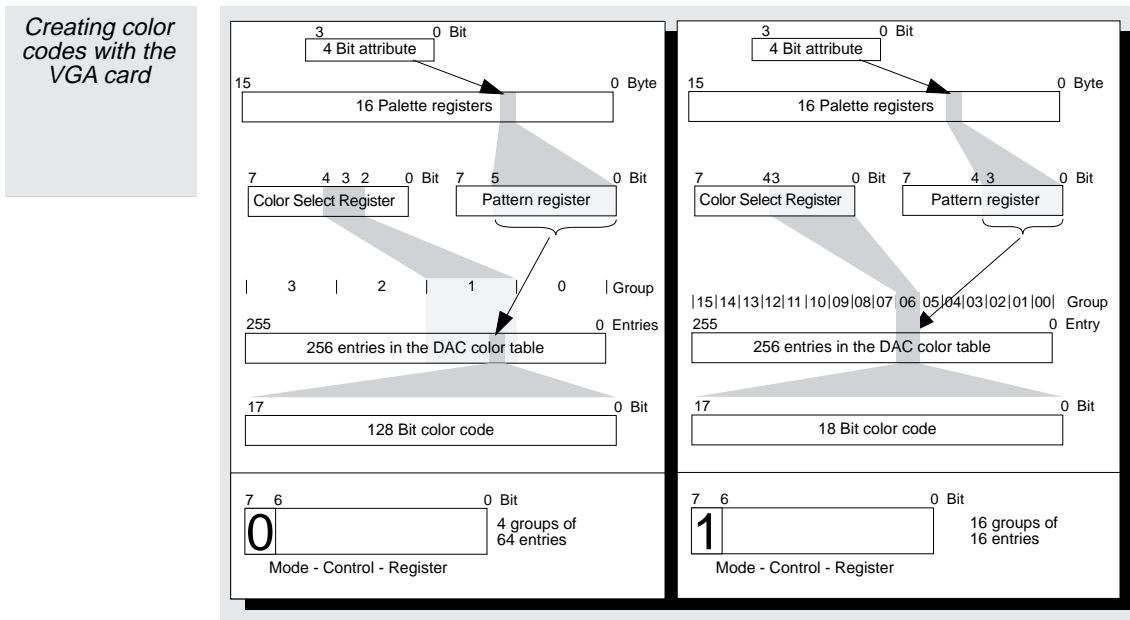
The DAC color table

The palette register contents can be passed directly from the EGA card to the monitor, through the monitor cable's six color leads. However, this is impossible with the VGA card. The VGA card produces an analog signal for the monitor, which works under a completely different premise.

So, the contents of the VGA card's 16 palette registers are added to the DAC (Digital to Analog Converter) color table before color information can be sent to the monitor. As the name suggests, this table converts the digital color information from the palette registers to an analog signal the monitor can understand.

The DAC color table has 256 registers, each of which stores the information for one color selected from the total VGA palette of over 262,000 colors. This impressive number of colors is a result of 18-bit color coding ($2^8 = 262,144$). Each entry in the DAC color table consists of three 6-bit color values: one each for the red, green and blue color components.

To select a register in the DAC color table, the video controller interprets the contents of the palette register as an index to the DAC color table instead of as a color value. The following illustration shows how the contents of several other registers determine various ways of organizing the DAC color table into groups.



The mode control register of the video controller plays an important part in this process. If this register contains the value 0, then the index to the DAC color table is created from bits 0 through 5 of the corresponding palette register and bits 2 and 3 of the color select register. This means the DAC color table is divided into four groups of 64 consecutive registers and that bits 2 and 3 of the color select register determine which of the four groups is currently active.

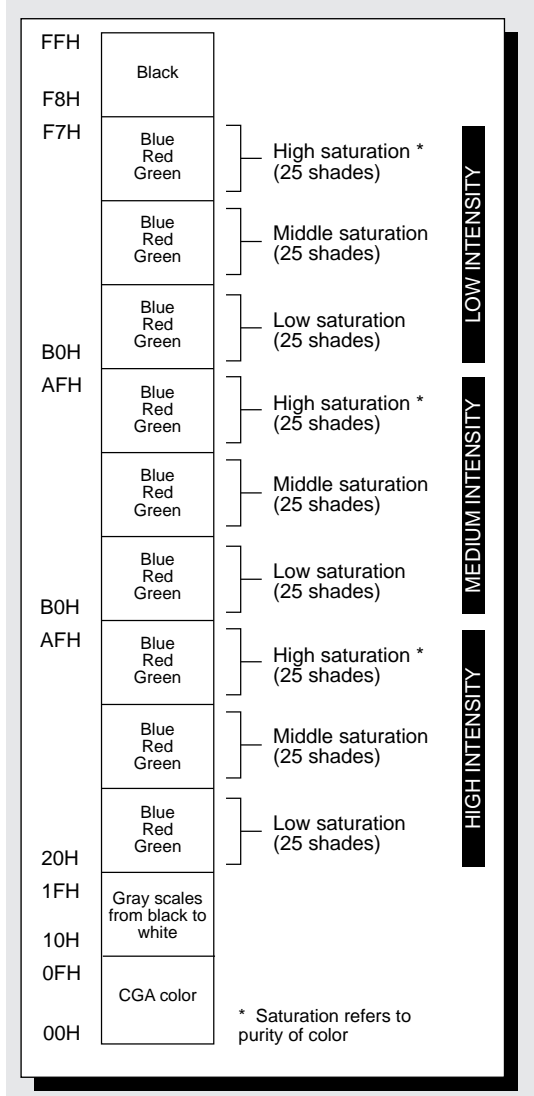
It's slightly different if bit 7 from the mode control register contains a value of 1. In this case, the DAC color table is divided into 16 groups of 16 consecutive registers. The index to the table is then created from bits 0 through 3 of the palette register and bits 0 through 3 from the color select register. Again, the value from the palette register is an index and the value in the color select register determines the currently active group in the DAC color table.

This type of coding can create fast and continuous color changes for entire groups of characters or pixels on screen. This is done by loading the color groups of the DAC color table with series of colors that have increasing or decreasing intensities. Then simply change the current DAC color group using the color select register.

To emulate default CGA colors, the VGA card is initialized so the 16 palette registers point to the first 16 registers of the DAC color table. These registers in the DAC color table are loaded with the color information for the standard CGA colors from black (0) to white (15). The other DAC color table registers aren't set when a text mode is initialized with video BIOS function 00H. In a graphics mode, all 256 registers of the DAC color table are initialized as long as the initialization isn't switched off by sub-function 31H of function 12H.

The following illustration shows the scheme used to initialize the DAC registers. There is also a short program at the end of this section that shows the initialization of the DAC registers on screen and allows you to make changes to individual registers.

*Initialization of
256 DAC color
table registers
(VGA graphics
modes)*



The palette registers themselves are the actual color sources in text and 16 color graphics modes. In the 256 color modes of the VGA card, however, this scheme would require 256 different palette registers. This is why the palette registers store index values to the DAC color table in these modes.

The 256 entries in the DAC color table determine the 256 different colors that can be displayed on the screen at any one time. So, programming the palette registers directly in these modes won't have the desired effect on the screen colors.

Setting colors using the BIOS

The expanded EGA/VGA BIOS has several functions for manipulating the contents of the palette registers and the DAC registers. The other register of the attribute controller can also be set. These tasks are accomplished with sub-functions of function 10H of the BIOS video interrupt. When calling one of these sub-functions, load function number 10H into the AH register and place the sub-function number in the AL register.

The first of these sub-functions is 00H, which enables you to load any color value into one of the 16 palette registers. To call this sub-function, load function number 10H into the AH register and the sub-function number 00H into the AL register. Also, load the palette register number (00H-0FH) into the BH register and the color number into the BL register.

The register number passed to this sub-function isn't checked. So, you can also use it to access the overscan register, which is found immediately after the last palette

register. Since this register determines the color of the screen frame and background color in the CGA compatible graphics modes, it also has its own sub-function (01H).

Since there are only two or three raster lines available for a screen frame, you should only use black as a background color, especially for the EGA text modes. Also, the contents of the overscan register are meaningless when a monochrome monitor is attached.

To call this function for accessing the overscan register, load the function number 10H into the AH register and sub-function number 01H into the AL register. Load the screen border color into the BH register. This is passed to the overscan register when the function is called.

Sub-function 02H loads both the palette registers and the overscan register in one operation. In addition to the usual function number and sub-function number, load the 17-byte address of a table that contains values for all 17 registers (16 palette registers plus the overscan register) into the ES:DX register pair. When the function executes, the 17 values from this table are loaded into the 17 registers.

Although we have two functions for changing the contents of the palette registers, the expanded EGA BIOS doesn't have any functions for reading the contents of these registers. The EGA card doesn't allow the contents of the attribute controller registers (and almost all other registers) to be read. This situation worsens when working with TSR programs because they aren't able to restore the palette registers to their original contents when the interrupted program is reactivated.

Many programs solve this problem by diverting the results of sub-functions 00H and 02H to a custom routine that stores the values in the palette registers before writing new values. This method doesn't work, however, if you attempt to bypass the BIOS functions and program the palette registers directly. So, you should always use the BIOS functions, even though Section 4.8.10 shows you how to manipulate the palette registers directly.

The last sub-function of function 10H in the expanded EGA BIOS defines the meaning of bit 7 in the attribute byte of a character in text mode. Just as with the CGA and MDA cards, this bit can be used with the EGA/CGA card to make a character blink or to display a character with an intense background color. The CGA and MDA cards require direct programming to define the meaning of this bit, but the EGA/VGA BIOS has a special function (sub-function 03H of function 10H).

Place the function number (10H) in the AH register and the sub-function number (03H) in the AL register. The BL register specifies the degree of intensity. Loading 0 into this register produces a high intensity background color and 1 enables character blinking when bit 7 of the attribute byte is set.

Other sub-functions of the expanded VGA BIOS

The VGA BIOS video interrupt has some functions that aren't included in the EGA BIOS. These apply to the DAC color table and palette register reading. Unlike the EGA card, the VGA card can easily read palette register contents. This also applies to several other registers that are inaccessible on the EGA card.

The contents of the DAC color registers can be modified with sub-function 10H. Place the function number (10H) in the AH register and the sub-function number (10H) in the AL register. Also, place the number of the desired DAC color register (0 - 255 [00H - FFH]) in the BX register and the desired color code in the CH, CL and DH registers. 18-bit VGA color codes consist of three 6-bit components, one for each color component (red, green and blue). Like many of the other sub-functions we'll describe, 10H expects the red component in DH, the green component in CH and the blue component in DL. Only the first six bits (0 - 5) are significant.

The DH, DL and CL registers are also used by sub-function 15H to return the contents of a DAC color register. Place the function number (10H) in the AH register and the sub-function number (15H) in the AL register. Also, place the DAC color register number in the BX register.

Sub-function 12H loads a number of DAC color registers in one operation. The BX register expects the number of the first DAC color register to be loaded and the CX register expects the total number of registers to load. Instead of processor registers, a buffer is used to pass the new values for the given DAC color registers. The address of this buffer is loaded in the ES:DX register pair. Each DAC color register receives three consecutive bytes from this buffer. In each group of three bytes, the first

byte provides the green component, the second byte provides the red component and the third byte provides the blue component.

Sub-function 17H allows you to read the contents of a range of DAC color registers. The number of the first register to read is loaded in the BX register and the total number of registers is loaded in the CX register. The VGA BIOS then copies the contents of these registers to the buffer with the segment and offset address specified in the ES:DX register pair. The structure of this buffer is the same as that described for sub-function 12H. Remember, each register from the DAC color table contains three bytes instead of one, so be sure that you have a large enough buffer.

You can determine the way the DAC color table is organized and which color group is active with sub-function 13H. This sub-function has two sub-functions of its own. If this function is passed the value 0 in the BL register, then bit 0 of the BH register is copied to bit 7 of the mode control register of the VGA controller and the DAC color table is divided into 4 or 16 groups. If the BL register contains the value 1, then the content of the BH register is copied to the color select register and the active color group of the DAC color table is selected.

The contents of these two registers can be determined by calling sub-function 1AH. After this function call, the BL register contains the contents of bit 7 of the mode control register and the BH register contains the contents of the color select register.

There is also a sub-function for converting the color codes of the DAC color table to gray scales. This is sub-function 1BH. This is helpful for displaying a black and white picture on a color VGA monitor. If you have a monochrome VGA monitor, the conversion of colors to gray scales takes place in the monitor itself.

To use sub-function 1BH, load the first register number into the BX register and the total number of registers to convert into the CX register. The actual conversion or gray scale summing, is done by weighting each color component to obtain a gray scale value between 0 (black) and 1 (white). The color component values are weighted so the red component is 30% of the final gray scale value, the green component is 59% and the blue component is 11%.

In addition to the selective conversion of the certain DAC registers to gray scales, you can use sub-function 33H of BIOS function 12H to convert the contents of the entire table. Before calling the BIOS video interrupt, the sub-function number is loaded in the AL register and the function number is loaded in the AH register as usual. In this case, the AL register determines whether the conversion occurs. A value of 0 tells the BIOS to convert the color values to gray scales. A value of 1 leaves the color values intact.

In addition to the sub-functions for manipulating the DAC color table, the VGA BIOS also has several sub-functions for reading the palette registers using function number 10H. Sub-function 07H reads the contents of any palette register. The number of the desired palette register is loaded in the BL register and its contents are returned to the BH register. The contents of the overscan register are also read with this sub-function, but the BIOS has also dedicated sub-functions 08H for this purpose. As with sub-function 07H, the result is returned to the BH register.

Sub-function 09H returns a copy of the contents of all 16 palette registers and the overscan register. This sub-function writes the contents of these registers to a 17-byte buffer. The segment address of this buffer is loaded into the ES register when the sub-function is called and the offset address is loaded into the DX register.

Sample programs

The BIOS functions make it easy to set the available colors. It's more difficult to select from these colors the actual colors your program will display on screen for a given character or pixel. This is especially true for the VGA card. The EGA card is limited to a choice of 64 colors, but the VGA's palette of 256 simultaneous colors makes this process more complicated. So, we'll conclude this section with a demo program called VDACC. The companion CD-ROM includes C and Pascal versions of this program called VDACC.C and VDACP.PAS.

The program works in the VGA 256 color graphics mode with a resolution of 320x400 pixels. The assembly language modules V3240PA.ASM and V3240CA.ASM, which were described in "The VGA 256 color graphics modes" in this section, are also used. Other routines, which you may remember from previous chapters, such as ISVGA, LINE and GRAFXPRINT, are also used.

The heart of the program consists of the routines SETDAC, GETDAC and DEMO. SETDAC and GETDAC are ports to sub-functions 12H and 17H of function 10H. They read and write any number of DAC color registers. The DEMO routine frequently uses these to load all the DAC registers and allows you to read or change the contents of individual registers.

On screen, you'll see a collection of 256 color blocks that are arranged in a square. Each color block consists of pixels of a certain color. The block in the upper-left corner starts with color value 0. The next block to the right uses color value 1 and its neighbor uses color value 2 and etc. to the lower-right block, which uses color value 255.

In this way, all 256 colors in the DAC color table are displayed on screen. First, you'll see the colors the BIOS automatically assigns when the graphics mode is initialized. In addition to the actual colors, the program also displays the numerical data for the color in the status line. When the program begins, the numbers of the red, green and blue color components for the upper-left color block will be displayed. You can then use the arrow keys to view the color component values for the other color blocks.

As you scroll through the color values, you'll notice the current color block is indicated by a white frame. The copyright message across the top of the screen will also be displayed in the color of the current color block. At first you cannot see the copyright message because it's initialized with the color black. To change the color of the copyright message, it is initialized with color code 255, which starts out as black. Then, each time you move the cursor, the color of the current block is copied to DAC color register number 255. This means the color of the lower-right color block and the color of the copyright message change with each cursor movement.

You'll find the following program(s) on the companion CD-ROM



VDACP.PAS (Pascal listing)
VDACC.C (C listing)

The program also allows you to change the red, green and blue components for the current color block with the **R**, **G** and **B** keys. Pressing one of these keys increments the value of the corresponding color component by one. The change is reflected immediately in the color of the color block, the color of the copyright message and the contents of the status line. To decrease the value of a color component, hold down the **Shift** key while pressing the corresponding letter. You can also press **Spacebar** to return a color to its original value. Press the **Enter** key to end the program. The program restores

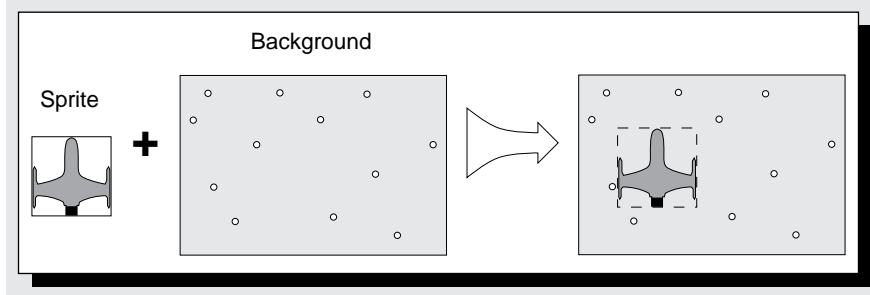
the original color table and returns you to text mode.

Sprites

Nothing dazzles computer users more than slick graphics. Whether it's a PacMan character zipping across the screen, a starship defending its homeworld from evil invaders or a dinosaur emerging from the jungle, good graphics and animation always catch the user's attention. However, behind every successful animation lie dozens of hours of development. This is especially true for PC software development because most home based PCs contain video hardware that has limited capabilities for graphics programming. So, a lot of coding must be done from scratch.

In this section, we'll show you how to create convincing graphics on the PC despite these limitations. The technique involves using sprites, which are graphics objects that are used in almost all computer games and many animation applications.

*Transparent
sprite display*



What are sprites?

A sprite is a rectangular block of pixels grouped together to form an object. You can then move this object across the screen. The colors used must be selected carefully. Many programmers assign a single color to the object and then assign the screen background color to all the pixels in the sprite block that don't represent the object itself. Although this simplifies development, it limits your animation options.

In addition to the basic image represented by the sprite, you should also be able to replace it with another image while the program is running. For example, the basic image may be a starship, but the second image may be an exploding starship.

Programming an application then involves coordinating the movement of sprites on the screen to create the desired animation. We won't discuss that here, but we'll cover all the routines you'll need for basic sprite programming. These include routines for sprite definition, movement, removal and image change.

The demonstration programs show examples of sprite programming on EGA and VGA cards. First, we'll look at the two VGA 256 color modes described in "The VGA 256 color graphics modes" in this section. Then, we'll explore the EGA/VGA 16 color modes with 640x350 pixel resolution. This mode is more difficult to program than the 256 color modes.

Each demonstration program exists in C and Pascal implementations. They all begin by filling the screen with the PC character set to create a background for sprite movements. Six sprites appear as spaceships, moving vertically on the screen. These ships bounce off the top and bottom screen borders, changing their appearances to point in a different direction when this occurs. Assembly language modules help increase the speed of sprite movement.

The two-page concept

Switching between two screens produces smooth animation. Sprite animation with only one screen page results in flickering as your objects move. This occurs because of the way sprites are displayed and moved on the screen.

There are several steps to this procedure. The screen area containing the sprite is processed several times. First, the background of the area on which the sprite is overlaid is saved for later restoration, after the sprite has moved to another location. This process is invisible to the user, because only a certain portion of the video RAM is being read.

In the next step, the sprite is copied to its location in video RAM and displayed on the screen. These steps themselves don't cause screen flickering. The critical point is when you move the sprite again to create the illusion of motion. In this case, the background of the sprite must first be restored by copying the old pixels from the buffer back to video RAM.

Although this process is fast, it's visible on the screen. The sprite disappears momentarily and then reappears in its new location. Then the entire cycle of saving the background, writing the sprite and restoring the old background is repeated.

The flickering effect cannot be avoided if you're working with only a single screen page. This is why we use two screen pages when programming sprites; this allows you to switch smoothly between screens. The processing always occurs on the hidden screen. When the processing is complete, the next scene in the animation is displayed and the cycle begins again.

The need for two screens also explains why we don't use the high resolution 640x480 VGA mode. A single screen page in this mode requires 150K, leaving only 106K. So, we limit the resolution to 640x350 in the EGA and VGA 16 color modes. This mode is adequate for most games, which are the most common applications for sprites.

Regardless of the video mode and how we choose to write sprites in video RAM and store their old backgrounds, there are certain universal problems that we must address when programming with sprites. The solutions to these problems are similar in both the C and Pascal versions of the demonstration programs. So, we'll discuss common aspects of the two versions before viewing the differences. These occur primarily in the modules that address video RAM, which is organized differently for the various graphics modes. These modules can be modified so the demonstration programs work with other types of video cards or even in the graphics modes of the Super VGA card.

Structure of the sprite programs

The sprite demo begins by determining the video card type. Next, a routine named DEMO is called from the main program or the MAIN() function. The DEMO routine fills screen pages 0 and 1 with characters from the PC character set using the

GRAFXPRINTF (C version) and PRINTCHAR (Pascal version) routines (see "The 16 color EGA and VGA graphics modes" and "The VGA 256 color graphics modes" in this section for descriptions of these routines). These routines in turn access assembler modules, such as V3220CA.ASM and V3240PA.ASM to enable various video modes (see "The VGA 256 color graphics modes" in this section). The assembler modules contain routines for reading and writing pixels, switching between and displaying screen pages and initializing a specific graphics mode.

After the characters form the background for the sprite display, a copyright message appears in the middle of the screen. Then the COMPILESPRITE routine defines the sprites.

The sprites aren't actually created by COMPILESPRITE. A routine called CREATESPITE later creates the sprites in each program. A string array conveys the appearance of a sprite. The COMPILESPRITE routine converts this array to a binary format that is later used to display the sprite on the screen. COMPILESPRITE begins by creating a bit pattern that can later be assigned to a number of sprites. COMPILESPRITE accepts a different number of parameters, depending on the version of the program. The first two parameters are always the same, however. The first parameter contains the string array that describes the appearance of the sprite and the second parameter contains the number of strings in the array.

This second parameter also defines the height of the sprite. A sprite can have a height ranging from one to hundreds of pixels. Each pixel line is represented by one string in the array. The first string is the top line of the sprite, the next string is the second line from the top, etc. Each character in a string represents a pixel. Since a sprite is a rectangular object, all strings in the string array have identical widths. So it's unnecessary to pass the width of the sprite, since this can be obtained from the width of the first string in the array.

The following is an example of how a sprite may be coded in a C array:

```
static char *STARSHIPUP[20] =
{
    "          AA          ",
    "          AAAA         ",
    "          AAAA         ",
    "          AA           ",
    "          GGBBGG        ",
    "          GBBCCBBG       ",
    "          GBBBCCBBBG      ",
    "          GBBBBBBBBBBG    ",
    "          GBBBBBBBBBBG    ",
    " G          GBBBBBBBBBBBG  G ",
    "GCG          GGBBBBBBBBBBDGG  GCG",
    "GCG  GGBBDBBB  BBDBBBGG  GCG",
    "GCBGGGBBBBBBDB  BBDBBBBGGGBCG",
    "GCBBBBBBBBBBDB  BDBBBBBBBBBBCG",
    "BBBBBBBBBBBBBDB BB BDBBBBBBBBBBB",
    "GGCBBBBBBDBBBBBBBBBBDBBBBBBCG",
    "  GGCCBBDDDDDDDDDDDDDBBBCCG",
    "    GGBBDDDDGGGGGDDDDDBBG",
    "      GDDDGGG  GGDDDDG",
    "        DDDD      DDDD"
};
```

The sprite demonstration programs were intended for use with color monitors, so each pixel must have a color associated with it instead of a simple "on/off" status. The programs read the @ character as black (color code 0), A as blue (color code 1), B as green (color code 2) and so on.

Pixels that aren't associated with the sprite appear in the background color (blank spaces).

The COMPILESPRITE routine; always returns a pointer to a structure of type SPLOOK, which is placed on the heap by the COMPILESPRITE routine. This structure contains all the important information needed about the sprite. Sprite creation

requires this structure in the following steps. After `COMPILESPRITE` is finished, the string array initially used to define the sprite is no longer needed.

Two types of sprites are defined within the programs. They are represented by the string arrays `STARSHIPUP` and `STARSHIPDOWN`. One sprite displays the ship going up and the other displays it going down. Using the pointer to the `SPLOOK` structure, the sprites are generated within a program loop.

The `SPRNUM` constant controls the number of times the loop is processed, which is set to a value of 6 in each demonstration program. Each time the loop is processed, a new sprite is created with a call to the `CREATESPRITE` routine. If you want to experiment with the program, you can easily set this constant to a higher value to create more sprites on the screen. The sprites will be crowded closer and closer together until they overlap. This will cause problems with your display, since these demonstration programs aren't written to handle collisions between sprites.

When experimenting with the number of sprites displayed on screen, you'll notice the sprite movement suddenly becomes jerky and hesitant. Reduce the number of sprites by one until smooth movement returns. The number of sprites that can be smoothly processed depends on the processor speed and graphics card type in your system. The reason for this problem will become clear as we discuss sprite movement in more detail.

The `CREATESPRITE` routine creates sprites through recursive calls. The first parameter required by this routine is the pointer to the sprite description, as returned by `COMPILESPRITE`. This is how the size and appearance of the sprite are defined. The rest of the parameters passed to `CREATESPRITE` depend on the sprite program and will be discussed along with the individual details of each program as we encounter these details.

In all the sprite demonstration programs, the `CREATESPRITE` routine returns a pointer to a structure of type `SPID`. This structure stores all the relevant data for the sprite that was just created. This information includes the pointer to the appearance of the sprite, its current position in both screen pages and other information that will vary with the program version. This data structure is also placed on the heap, just as with `COMPILESPRITE`.

The loop used to create sprites also determines their initial position on screen and the speed at which they will move. This information is generated by a random number function. The speed of movement in the X direction is always set to 0 in the local variable `DX`, which ensures the sprites will move only up or down, but never right or left. When experimenting with these programs, you may want to set this variable to a value other than 0 and watch what happens to the sprites.

The pointer to the sprite (or its `SPID` structure) and its speed of movement are stored in a local variable called `SPRITES`. This variable is a simple array.

The `SETSPRITE` procedure displays the sprite on the screen. The first parameter required is the pointer to the sprite description that was returned by `CREATESPRITE`. Then the X- and Y-coordinates for the sprite in screen page 0 are passed, followed by the X- and Y-coordinates in screen page 1.

The two coordinate pairs are passed separately because they cannot be identical. This is the basis of the two-page concept. If the sprite were located at the same position in both screen pages, it would not appear to move at all when you switch pages. The speed of the sprite's movement is determined by the distance between the sprite's positions in the two screen pages.

Independent movement

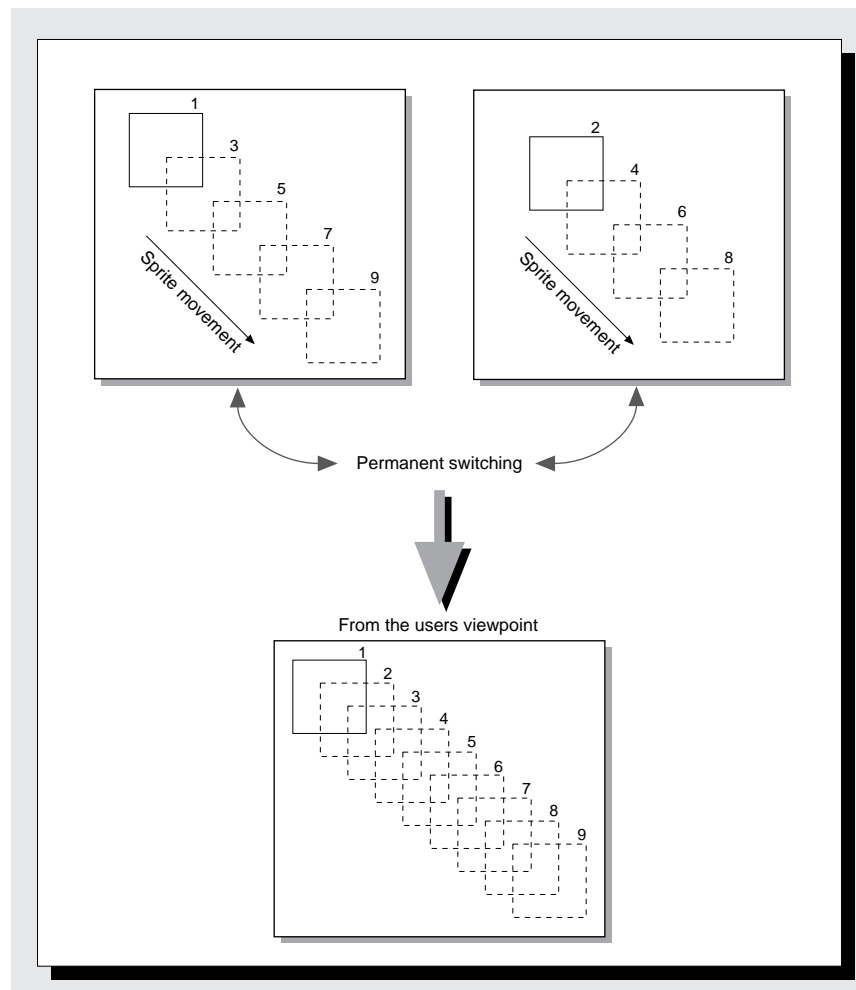
Imagine a sprite that you want to move vertically on the screen from the top edge (Y-coordinate = 0) to the bottom edge. Each time the screen is redrawn, the sprite moves down by one pixel. We can initialize the program with the sprite at Y-coordinate 0 in the first screen page and at Y-coordinate 1 in the second page. Then when the display switches from page 0 to page 1, the sprite appears to move down one pixel.

Now, while screen page 1 is being displayed, the sprite in screen page 0 is moved to a new location. The new location will be two pixels below its original position, so when screen page 0 reappears, the sprite appears at Y-coordinate 2 (one pixel below its previously displayed position). This means that if you want the sprite to appear to move by a certain number of pixels when each new screen is drawn, you must move it by twice as many pixels when internally building a new screen page.

The same is also true for changes in the X-coordinate to make your object move horizontally.

After the sprites are created and initially displayed, a loop processes their movements until the user presses any key on the keyboard. Inside the loop, the program constantly switches between the two screen pages and the sprite is moved on the screen page that isn't displayed.

Sprite display and movement in the two screen pages



This is accomplished with the MOVESPRITE function. This is one of the few sprite routines that appears unchanged in all the sprite demonstration programs. That is because this routine is based on the lower level routines that reflect the actual differences between the programs.

The arguments for this routine are a pointer to the sprite description, the page number in which you want to move the sprite and the number of pixels to move in the X- and Y-directions. The current sprite position and the movement increment ("speed") calculate the new position. Collisions with the edge of the screen are also taken into consideration. If the new sprite position is different from the old position, the movement executes.

The RESTORESPRITEBG routine; deletes the sprite from its current position by restoring the background. The background of the sprite's new location is stored in a buffer using the GETSPRITEBG routine. Finally, the sprite itself is copied to its new location using the PRINTSPRITE routine.

The result of the MOVESPRITE function is a byte that reflects any collisions with the edge of the screen. The constants OUT_LEFT, OUT_TOP, OUT_RIGHT, OUT_BOTTOM and OUT_NO can determine whether any collisions have occurred.

The DEMO routine also uses the information in this byte. Remember, when the sprite collides with the edge of the screen, it changes its appearance as well as its direction. This keeps the nose of the ship pointing in the correct direction.

The execution speed of the sprite movement loop within DEMO (and the speed of the sprite on screen) varies with your processor's speed, your video card and the bus that connects the two. The frequency of your screen picture also makes a difference because the rate at which screen pages are changed to create movement must be synchronized with the picture frequency.

A query loop within the assembler routine SHOWPAGE handles this. SHOWPAGE returns to its caller after a screen redraw has begun. The new screen page is selected and displayed only after this occurs. This prevents the caller of the SHOWPAGE routine from accessing a screen page that is still visible for a short time because the next screen redraw hasn't begun yet.

Because of this, it's possible that up to 1/50 of a second is lost, depending on the picture frequency and whether SHOWPAGE is called immediately after a screen redraw has begun. Once you've accumulated a certain number of sprites on the screen, it starts taking too long to switch screen pages. This shows up on the screen as a hesitation in the movement of the sprites.

The speed at which the sprites appear to move is determined by the number of pixels the sprite moves each time the movement loop is processed. There are practical limits here. If you move the sprite too far in a single screen change, it will appear to jump and the impression of fluid movement is lost. The speed with which your system can process the movement loop is also a determining factor.

To limit these problems, the most important routines that connect the program and the video card are written in assembly language. These assembler routines are called by the routines GETVIDEO and PUTVIDEO, which in turn are called by high level language routines, such as PRINTSPRITE, GETSPRITEBG and RESTORESPRITEBG.

PUTVIDEO and GETVIDEO store an area of the screen in a buffer or fill a screen area with a pixel pattern that has already been stored in a buffer. These routines are used by PRINTSPRITE, GETSPRITEBG and RESTORESPRITEBG to display a sprite on screen, save a background area and restore a background area.

The assembly language routines called by GETVIDEO and PUTVIDEO differ significantly depending on the graphics mode being used. In the next section, we'll focus on these routines when we discuss the differences caused by the various graphics modes. We'll start with the VGA 256 color modes. This mode isn't available on the EGA card, but the program is actually simpler in this mode than in the 16 color modes. A discussion of the 640x350 pixel mode for both EGA and VGA appears at the end of this section.

Sprites with 320x200 pixels and 256 colors

Graphics programming requires many compromises. For example, although the 320x200 pixel mode with 256 colors has low resolution, the numerous colors and the ability to program with four screen pages can be great advantages for many applications. Although only two screen pages are needed to animate with sprites, the other 128K of video RAM can be used to store screen backgrounds and bit patterns for sprites.

This is not just a matter of using less of the main memory. Keeping this information in video RAM can significantly increase the execution speed of your program. Remember, only one byte can be passed from the bitplanes to main memory at once, so this also taxes the system bus. Such limitations are removed if you move these memory areas around exclusively within video RAM.

Not only can we avoid I/O operations between the CPU and the video card over the system bus, but we can also enable four bytes to be copied using the four latch registers of the VGA card at once with a single MOVSB command. The REP MOVSB assembly language instruction allows an entire graphics line to be copied in one operation. This is a sufficient reason to put the unused portion of the video RAM to work doing things besides storing complete screen pages. Unfortunately, there is a problem with this procedure, as we'll see later.

First, let's discuss the programs. In the C version, the modules S3220C.C, S3220CA.ASM and V3220CA.ASM generate sprites in the 320x200 pixel 256 color mode. We've already seen V3220CA.ASM in "The VGA 256 color graphics modes" in this section. The S3220C.C and S3220CA.ASM modules are new. The S3220C.C module contains the C routines COMPILESPRITE through GETVIDEO, which we've already discussed.

The assembly language module 3220CA.ASM contains only the BLOCKMOVE routine, which moves a rectangular block of pixels within video RAM. The Pascal version isn't much different. The modules are called S3220P.PAS, S3220PA.ASM and V3220PA.ASM. V3220PA.ASM was also discussed in "The VGA 256 color graphics modes" in this section.

Storing sprite information in video RAM itself allows us to move this information quickly to different locations. But the structure of video RAM in the 320x200 pixel mode imposes certain limitations on your program. The width of each sprite must always be rounded to a multiple of four. This is because four pixels (i.e., four bytes) are always copied simultaneously using the four latch registers.

If the sprite width isn't a multiple of four, then you would always have to ensure that you copied only the bytes belonging to the current line. This means one or more latch registers would have to be excluded from the copy operation of the last group of bytes in a line. We always round the width of each sprite to a multiple of four, if only to save some development time.

There's another issue involved in those multiples of four. If the information for the sprite's appearance begins at an X-coordinate that is a multiple of four, then it can only be copied to another X-coordinate that is a multiple of four. A sprite starting at X-coordinate 0 can only be copied to coordinates such as 4, 8, 96, 224, etc., but never to coordinates such as 1, 2, 5, 13 or 182. This is simply because any read access to the video RAM in 320x200 pixel mode must begin with an X-coordinate that is a multiple of four. You must do this if you want to use all four latch registers together. The same is true if you would like to write data with all four latch registers in one operation.

There is only one way to move the copies of the four latch registers one pixel to the right, to X-coordinates such as 1, 2 or 25. You cannot simply move the contents from one latch register to the next. For example, you cannot move latch register 0 to latch register 1 or latch register 1 to 2 or 2 to 3, etc. Likewise, you cannot save the contents of latch register 3 and write it to latch register 0 as part of the next copy operation. However, it's possible to mask latch register 0 before the write operation, so its contents aren't copied to an X-coordinate in video RAM that is divisible by four.

Even this last possibility would require too much effort. So instead of this, we simply store four copies of the sprite with the first one starting at an X-coordinate that is a multiple of four. The second copy is stored starting with the next pixel to the right. This keeps the far left pixel column of the sprite unused. For now, this column takes on the background color and is seen as transparent. The third and fourth sprites are stored in the same way, with two or three pixel columns remaining transparent.

Sprite definition in video RAM: 320x200 pixel mode with 256 colors

Before a sprite can be displayed on the screen, a sprite must first be moved to an X-coordinate that is divisible by four. The difference between the starting X-coordinate and the target X-coordinate equals the number of empty columns at the start of the sprite. This also gives the number that identifies the sprite to be copied. The following formula can be used:

```
TargetX = int(X / 4) * 4
[or] X and not(4)
Sprite = X - int(X / 4) * 4
[or] X and 3
```

However, the problem of the transparent pixel columns still exists. As long as you use the four latch registers together and copy in groups of four pixels, the complete sprite is copied exactly as it was stored and the blank columns will remain blank. So before the write operation we must use the map mask register of the sequencer controller to mask the latch registers that would otherwise be overwritten with background pixels.

Since there is a different latch register for each group of sprite pixels, the map mask register must be specially programmed for each case before writing to the latch registers. This is done using a value calculated while the sprite description was being compiled with COMPILESPRITE. This value is passed to the BLOCKMOVE routine as part of an array.

Although this makes the procedure more complicated, it's unavoidable if you want to store the sprites directly in video RAM. Fortunately, this is only required when writing the sprite itself. It's not necessary to program the map mask register when saving or restoring a sprite background.

This method of storing sprites in video RAM and copying them as part of an array containing values for the map mask register plays a major role in the S3220C.C and S3220P.PAS modules, as well as the BLOCKMOVE procedure; used in the S3220CA.ASM and S3220PA.ASM assembly language modules.

This starts with the COMPILESPRITE routine. The string array that defines the sprite and the sprite's height are the first two parameters passed to this routine. There are four others. The first of these specifies the screen page that is used to store the sprite description. Since pages 0 and 1 are being used for the display, this must be either 2 or 3.

The next parameter is the pixel line in which the sprite definition will begin. This can be any value from 0 to 200 (the sprite height). Be careful not to overlap sprite descriptions in video RAM.

Within the given line, the four copies of the sprite are stored right next to one another. You can actually see what this looks like by displaying the screen page with SHOWPAGE after compiling the sprite with COMPILESPRITE in the DEMO routine.

The next parameter for COMPILESPRITE is a character that represents the smallest color value in the string array used to define the sprite. Usually, you use the letter 'A' for this. Of course, you can also use lowercase letters or numbers to code the pixel colors within the string array, in which case you would enter 'a' or '0' for this parameter.

The last parameter also handles pixel colors. It gives the color number assigned to the sprite pixel with the smallest color code ('A', 'a' or '0'). This defines the meanings of all the other characters in the string array, since 'B', 'b' or '1' corresponds to the next color value in the palette.

In this way, more than 128 colors can also be reached without using foreign or special characters in your string array. Regardless of the values in the last two parameters, an empty space always represents a transparent pixel. When the sprite is displayed, the transparent pixel turns to the background color.

COMPILESPRITE uses the information passed to it to build the sprite four times in the given page, as previously described. Background pixels are assigned color code 255 so they can be distinguished from the non-transparent pixels in what follows.

Next, the four sprites are processed again to fill the array that is later used to program the map mask register when BLOCKMOVE is called. The memory for this array is allocated on the heap. Each pixel in the four sprites gets a nibble containing the corresponding value for the map mask register.

The pointer to this array is stored in the sprite description (SPLOOK) along with all other relevant information. A pointer to this structure is returned to the calling routine.

CREATESPRITE also requires more information. In addition to the obligatory pointer to the sprite description, the locations of the two areas in video RAM used to store sprite backgrounds from screen pages 0 and 1 must be passed. This requires the screen page (2 or 3) and the X- and Y-coordinates. Remember, you need an area twice as wide as the sprite area itself. This is because the two buffers from both page 0 and page 1 are stored next to each other.

Once a sprite is created with this procedure, you can use the SETSPRITE routine to display it on screen and MOVESPRITE to animate it. PRINTSPRITE, GETSPRITEBG and RESTORESPRITEBG support the BLOCKMOVE assembly language routine.

BLOCKMOVE expects a number of parameters, including the starting coordinates of the source and target areas. These locations are represented by the combination of the screen page plus the X- and Y-coordinates. The width and height of the rectangular block are also required. Finally, BLOCKMOVE receives a pointer to the array that contains the values for programming the map mask register.

If all pixels in the rectangular sprite should be copied regardless of the background pixels, then the array isn't required. You can pass a NULL pointer for this parameter. In the Pascal version, this is represented by the predefined NIL constant and in the C version by the NOBIT MASK constant.

If BLOCKMOVE encounters a NULL pointer, it copies an entire pixel line from the specified area in one move. This is much faster than the normal copy routine, in which the map mask register is programmed before each four byte transfer.

In either case, write mode 1 is set before the copy loop begins. Only this mode allows simultaneous transfer of the latch registers' contents to the four bitplanes. The routine restores the original write mode and ends.

Sprite programs using the 256 color 320x200 pixel graphics mode run faster than the other demonstration programs in this section because they can use video RAM for storing sprite descriptions and sprite backgrounds. Unfortunately, a screen resolution that is lower than what PC users are accustomed to is used. We believe that if 256 colors are needed, the 320x400 pixel mode is better, (even despite the slower execution speed).

You'll find the following program(s) on the companion CD-ROM



S3220P.PAS (Pascal listing)
S3220PA.ASM (Assembler listing)
S3220C.C (C listing)
S3220CA.ASM (Assembler listing)

Sprites in 320x400 pixel mode with 256 colors

When using the higher resolution 320x400 pixel graphics mode, we must sacrifice the advantage of sprite background and sprite description storage in video RAM. Instead, we must move this information from conventional memory to video RAM. These routines take longer to execute than those used to copy structures directly within the video RAM, but the user shouldn't notice the change in speed.

However, this mode simplifies creating and displaying sprites because the transfer from conventional RAM to video RAM occurs one byte at a time. We can also remove our artificial limitation of making sprite widths divisible by four and the need for maintaining four copies of each sprite. So, a sprite of any width can be copied to any X-coordinate. Also, this mode eliminates the need for the map mask register.

The different method for storing sprite information and for moving the information to video RAM can be seen in the sprite modules required for programming in this mode. These are S3240C.C, S3240CA.ASM and V3240CA.ASM in the C version and S3240P.PAS, S3240PA.ASM and V3240PA.ASM in the Pascal version.

The PUTVIDEO and GETVIDEO routines used by the modules previously listed contain the interface between conventional RAM and video RAM. GETVIDEO transfers the contents of a rectangular screen area from video RAM to conventional RAM, as in cases where you would save a sprite background to a buffer. PUTVIDEO then restores an area saved by GETVIDEO from the conventional RAM buffer to a specified location in video RAM.

These routines are based on two routines called COPYBUF2PLANE and COPYPLANE2BUF from the S3240CA.ASM and S3240PA.ASM assembly language modules. These routines transfer a rectangular area of pixels either from a specified bitplane to conventional RAM or from conventional RAM to a bitplane.

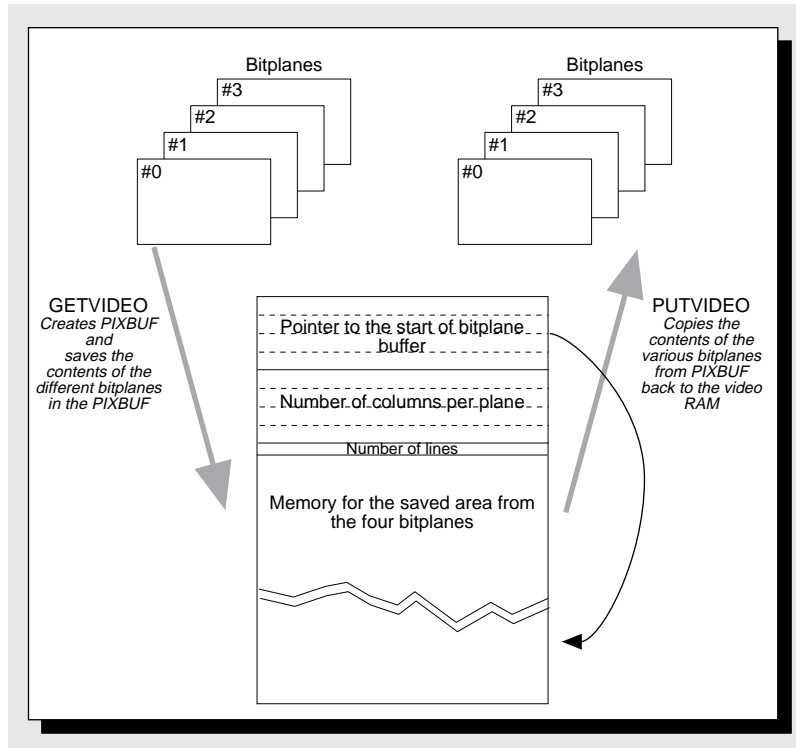
The four bitplanes, used to store pixels in all 256 color modes, are each handled separately. This is why these routines are called four times when they are used within GETVIDEO or PUTVIDEO. This works faster than trying to take four pixels (one from each bitplane) in a single move, in which case the read map register of the graphics controller or the map mask register of the sequencer controller would also have to be programmed before each move. Using the method previously described, these registers need to be programmed only once with each COPYBUF2PLANE or COPYPLANE2BUF call, because these routines access only one bitplane at a time.

GETVIDEO and PUTVIDEO are able to work with rectangular blocks of any width, not just multiples of four. So, the areas to be processed within each bitplane may not always be the same width. Imagine a GETVIDEO call where you want to load an area from video RAM with a width that extends from the X-coordinates 0 to 6. In this area, each pixel line has two pixels from the first bitplane (at X-coordinates 0 and 4), two pixels from the second bitplane (at 1 and 5) and two pixels from the third (at 2 and 6). The fourth bitplane therefore contains only one pixel, the one at X-coordinate 3.

Since GETVIDEO is called separately for each bitplane, the routine must keep track of how many pixels are involved and other information that is needed when GETVIDEO is recalled. This information is stored in a data block allocated on the heap by GETVIDEO when it is called. A pointer to this data block is passed back to the caller.

The data block is of type PIXBUF, which begins with an array of four pointers that point to the four buffers that contain the screen areas represented by the four bitplanes. This is followed by another array with four entries that indicate the number of pixels per bitplane. The last entry in this structure is the height of the screen area or the number of pixel lines.

Saving screen areas with GETVIDEO and PUTVIDEO in this mode



The screen area's height is the last entry in the PIXBUF structure. GETVIDEO places the buffer, which stores the pixel information from the individual bitplanes, after this structure. Space for this additional buffer is created when GETVIDEO allocates the PIXBUF buffer. The buffer size required for the information from the four bitplanes is added directly to the size of the PIXBUF structure. The result is a block of memory containing the PIXBUF structure, followed by the pixel information from the four bitplanes as loaded by COPYPLANE2VIDEO.

GETVIDEO returns a pointer to its caller, pointing to the pixel buffer. This pointer can then be passed on to subsequent PUTVIDEO calls. This allows the saved screen area to be copied to any position in video RAM, as often as desired.

The GETVIDEO function serves another purpose. If GETVIDEO is passed a pointer as its last parameter with a value other than NIL (in Pascal) or the ALLOCBUF constant (in C), then the information is loaded into an existing buffer (indicated by the pointer) that was created in a previous call to GETVIDEO. This saves time by allowing you to reuse buffers without having to create a new buffer on the heap each time you call GETVIDEO. When doing this, you must be sure the new screen area to be saved fits within the existing buffer.

The sprite demo programs for the 320x400 pixel mode use this feature for tasks such as saving a sprite background. This works because the size of a given sprite and consequently the size of its background, remains constant, so the pixel buffer needs to be allocated only once.

GETVIDEO and PUTVIDEO aren't only used for saving and restoring sprite backgrounds. These routines also compile the sprite description in COMPILESPRITE. Using the information in the string array that defines the sprite, this routine builds the sprite at screen coordinates 0/0 in the specified screen page (the screen page is passed as a parameter).

Then the area occupied by the sprite is simply copied to a pixel buffer by calling GETVIDEO. Any time you want to display the pixel on screen, simply pass this pixel buffer to the PUTVIDEO routine, with the screen page and the desired coordinates. The pointer to the pixel buffer is stored under the name PIXBP in the SPLOOK structure.

Sprite backgrounds are handled in a similar way. The CREATESPRITE routine uses GETVIDEO to create two pixel buffers of the same size as the corresponding sprite. The first buffer stores the sprite background from the first screen page and the other buffer stores the background from the second page. The pointers to these two pixel buffers are stored in an array called HGPTR within a structure of type SPID. CREATESPRITE configures this structure for identifying and working with sprites.

The pixel buffers for the sprite description and the two background areas are then used by routines such as PRINTSPRITE, GETSPRITEBG and RESTORESPRITEBG, to display the sprite and save or restore its background. You'll see these routines are simpler than those from the sprite demonstration programs for 320x200 pixel mode.

We haven't explained how background pixels are handled yet. The PUTVIDEO routine uses a Boolean variable called BG. This variable is passed on to the assembler routine COPYBUF2PLANE to determine whether the background should be taken into consideration when copying the buffer to each bitplane.

If this parameter contains the value TRUE, then COPYBUF2PLANE checks each byte to determine whether it represents a background pixel when copying the contents of the specified buffer to video RAM. Background pixels are assigned color code 255. This color code was assigned, within the CREATESPRITE routine, to all pixels represented by a space in the string array. If COPYBUF2PLANE encounters such a pixel, it skips it and doesn't write its color to video RAM. The routine then continues with the next pixel in the buffer.

This represents a rather simple solution to the problem of background pixels, even if it does add some time to the copy procedure between main memory and video RAM. However, the added time isn't really noticeable to the user.

Now we've covered the major points in the 320x400 pixel sprite programs. If you need additional information about how these demo programs work, the listings are fully documented.

You'll find the following program(s) on the companion CD-ROM



S3240P.PAS (Pascal listing)
S3240PA.ASM (Assembler listing)
S3240C.C (C listing)
S3240CA.ASM (Assembler listing)

Sprites in EGA and VGA 640x350 16 color graphics modes

If you'd rather have higher resolution at the expense of fewer colors, then you should use 16 color 640x350 pixel mode. Another advantage of this mode is that it's compatible with both EGA and VGA cards. The 640x480 pixel VGA mode cannot be used for sprite programming because it doesn't allow two screen pages to be stored in video RAM simultaneously.

The 640x350 pixel mode uses only 219K of the available 256K of video RAM for screen page storage. So 38,144 bytes remain; these bytes can be used for more than 76,000 additional pixels in a 640x120 pixel block. This extra video RAM can be used to store sprite descriptions and sprite backgrounds, as we did in the 320x200 pixel mode demonstration programs.

Depending on the number of sprites you define and the number of sprites actually displayed on screen, this extra video RAM can be used up rather quickly. As we'll see later, the main reason for this is that you must maintain eight copies of each sprite description instead of four.

Because of this potential limitation, the sprite demonstration programs in 640x350 pixel mode store their sprite descriptions and sprite backgrounds in conventional RAM. Although this is the same method used in the 320x400 pixel mode, the sprite creation and data transfer routines in these programs are quite different from the 320x400 mode programs. The main reason is the video RAM is organized in a completely different way for 16 color modes than for 256 color modes. We saw that eight pixels are represented by a single byte in 16 color modes, as opposed to one pixel per byte in 256 color modes. But one bit

isn't sufficient for representing a color code between 0 and 15. This is why the video RAM is organized into bitplanes, allowing the group of four corresponding bytes from each bitplane to define the color information for all eight pixels.

Organizing video RAM into bitplanes can make accessing a specific pixel slow and awkward. This affects the routines found in the S6435C.C (C version) and S6435P.PAS (Pascal version) modules, which display sprites on screen in the 640x350 pixel mode.

The sprite widths are rounded to multiples of eight pixels. If you don't do this, you must isolate only those desired bits from the last byte, save them and then write them back to ensure that only those bits belonging to the sprite are copied. This also applies to programming the map mask register, which is an alternative method for allowing sprites of any width.

This rounding is done internally and is transparent to routines, such as COMPILESPRITE, CREATESPITE and MOVESPRITE. The fact that eight copies of the sprite description are kept is also transparent. Eight copies are required because of a problem similar to that found in 320x200 pixel mode.

For example, if the binary coding of a sprite description begins in bit 7 of the first byte, it can be copied only to a screen area with an X-coordinate that is a multiple of eight if you want the description to remain unchanged. If you copy to any screen area that starts at an X-coordinate whose value mod 8 is unequal to zero, you must then shift all bits to the right by a value equal to the result of the mod 8 operation. This would actually take too much time to execute.

So, the COMPILESPRITE routine used in S6435C.C and S6435P.PAS creates eight copies of each sprite description, where each copy is shifted one bit to the right. Whenever you must move a sprite to a certain screen position, simply select the copy that matches the desired X-coordinate.

The eight copies of a given sprite description are stored in the form of a pixel buffer. Just as we saw in the 320x400 pixel demonstrations, the sprite description is built in video RAM and then loaded into a pixel buffer with GETVIDEO. Unlike the 320x400 mode, however, this operation is repeated eight times instead of four. This requires much space in conventional RAM.

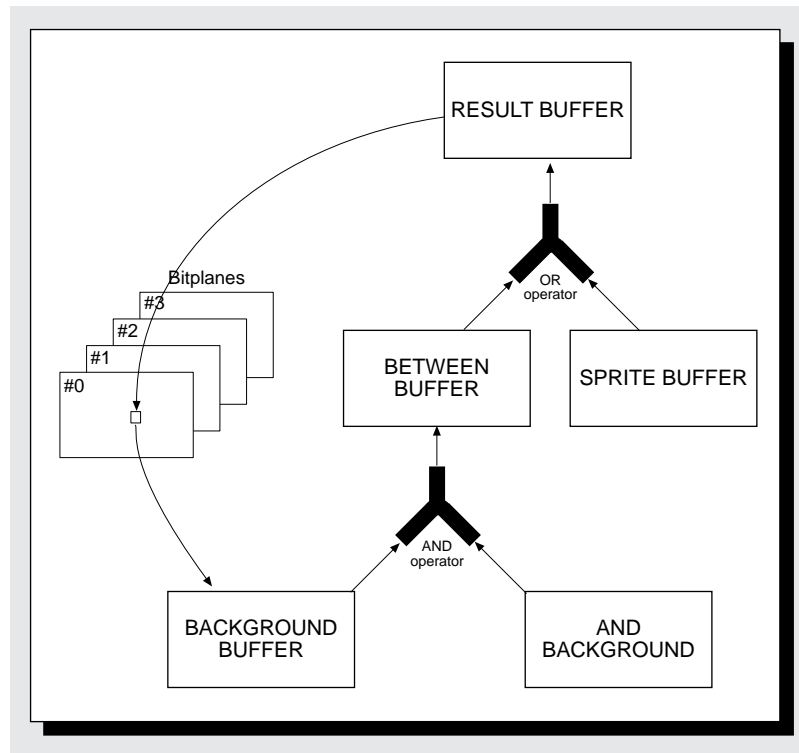
For example, imagine a sprite consisting of 20x30 pixels, which is a small object for such a high resolution screen mode. After you increase the width by seven pixels and round the width off to make it a multiple of eight, the result is 32 pixels:

```
int( ( 20 + 7 + 7 ) / 8 ) * 8 = 32
```

Each pixel requires four bits, so this sprite requires 16 bytes per pixel line. This amount multiplied by 20 equals 640 bytes for the pixel buffer, not including the status information in PIXBUF. This must be repeated eight times for each sprite description, which increases the number to 5K. When we include the two background buffers required for each sprite, we must add another 1K to the total.

An AND buffer must also be created for each sprite description. This makes the background transparent so the sprite doesn't appear as a rectangular block on the screen. Because of the way video RAM is organized in this mode, this isn't a simple task. It requires a three step process repeated for each byte to write a sprite to video RAM. This process involves loading the byte to be processed, programming the bit mask register to affect only the desired pixels and finally writing the value.

*Sprite
backgrounds,
sprite definition
and the AND
buffer*



Since the previous process would take too long to execute, you should use another method. Remember the sprite background is already available before we write the sprite to its new location. This means that we can merge the buffers containing the sprite description and the background information. This creates the desired pattern of background pixels and pixels before we access video RAM. This process results in the bytes that we actually want to write to video RAM, without having to mask certain bits.

This process involves taking the contents of the background buffer and changing only those bits that don't correspond to background pixels in the sprite description. This leaves the background pixels undisturbed and overwrites all other pixels with pixels from the sprite. This is accomplished with the AND mask.

Imagine a byte from the sprite description and the eight pixels it describes. Suppose that in this byte, the last pixel (represented by bit 7) should be transparent or part of the background. The AND mask for this byte would have all bits set to 0 except for bit 7, which would be set to 1. Then, when merging the buffers, the background byte combines with the corresponding byte from the AND buffer using a logical AND operation.

The result is that only the background bit, bit 7, remains unchanged. All other bits are set to 0. In the next step of the process, the result is combined with the corresponding byte from the sprite description using a logical OR operator. This byte contains a 0 in all bit positions that represent transparent background pixels. All other bits contain values that make up part of the four bit color code for a sprite pixel.

The result of this operation is that all pixels needed to display the sprite are changed to the colors given in the sprite description and all background pixels remain unchanged. This entire operation must be repeated for each bitplane. The AND mask is always the same, however, since the bit positions that represent sprite pixels and background pixels are the same for all bitplanes. The size of the AND buffer in bytes, which is only good for one of the eight copies of the sprite description, is:

$$\text{width} * \text{height} / 8$$

The AND buffers for each sprite description are created with COMPILESPRITE, in the same way as the sprite descriptions themselves. As the sprite array is processed, a 1 is placed in each bit that describes a transparent background pixel. A 0 represents a sprite pixel that overwrites its background.

The S6435CA.ASM and S6435PA.ASM assembly language modules perform the combination of the sprite background, the sprite description and the AND buffer. This routine is called MERGEANDCOPYBUF2VIDEO. As its name suggests, this routine merges the necessary buffers to create the desired pixel pattern and copies the buffers from conventional RAM to video RAM. Actually, these operations are done in parallel, byte by byte. First, a byte from the three buffers is merged, then it's copied to video RAM; the process continues with the next byte.

This process repeated four times in MERGEANDCOPYBUF2VIDEO for each of the four bitplanes. This also applies to the COPYBUF2VIDEO and COPYVIDEO2BUF assembly language routines which appear instead of the COPYBUF2PLANE and COPYVIDEO2PLANE routines which are found in the assembler module used in 320x400 pixel mode.

These routines can process all four bitplanes automatically because the width of the screen area is limited to multiples of eight

You'll find the following program(s) on the companion CD-ROM



S6435P.PAS (Pascal listing)
S6435PA.ASM (Assembler listing)
S6435C.C (C listing)
S6435CA.ASM (Assembler listing)

in this mode. So, it's unnecessary for the calling routine to specify the number of bytes to process. So COPYVIDEO2BUF copies the given screen area from all four bitplanes to the specified buffer and COPYBUF2VIDEO copies the specified buffer's contents to all four bitplanes.

This concludes our discussion of conceptual differences in the 640x350 mode sprite demonstration programs. Many features have been documented in the earlier programs in this chapter.

EGA and VGA card registers

EGA and VGA cards are based on several different controllers that share the work of generating a video signal. These are the CRT controller, the attribute controller, the graphics controller, the sequencer controller and the digital to analog controller (DAC), which is found only on VGA cards. In addition to these, EGA and VGA cards use several general registers. We'll discuss these controllers and registers in this section.

Originally, you could actually see which component on the card was responsible for which function because each was performed by a different chip. However, because of advancements in computer chip technology, all functions are now handled by one or two integrated controller chips.

In many instances, these integrated chips perform functions and graphics modes that far exceed the original EGA/VGA standard. To maintain compatibility, however, the register assignments are kept as similar to the original IBM EGA and VGA registers as possible. You'll find the registers described in this section on almost all EGA/VGA cards. Any registers that have different functions in EGA and VGA cards are discussed in detail.

WARNING

You should not change some registers. The most important of which are the CRT controller registers that manage the video signal and synchronize horizontal and vertical rescans. The interaction of these registers is very complex and improper programming could cause monitor damage.

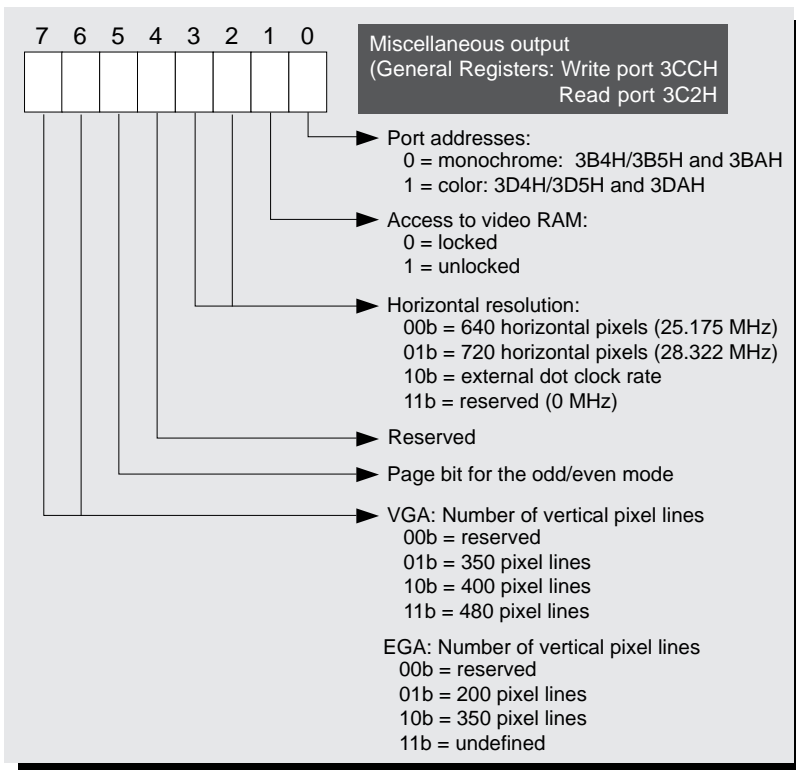
There are many other bits and registers that you can manipulate without problems. However, you should be careful because sometimes there are small but important differences between the EGA and VGA cards. We'll document these differences in this section when possible.

This section doesn't cover registers that don't follow the EGA/VGA standard. Once we look past standard EGA and VGA functions and registers, standards don't exist. For example, there are no register standards for Super EGA or Super VGA. The three most popular Super VGA cards differ significantly in register assignments and expanded functions.

The registers we'll examine here show how many options haven't been explored yet in the expanded EGA/VGA BIOS.

General registers

Besides the special controller registers, other EGA and VGA registers transfer more general information, which is used in the operation of the card. These are called the *general registers*.



0 For MDA card emulation, this bit specifies the port address of the CRT data and index registers and the input status register 1. These registers normally occupy ports 3D4H/3D5H and 3DAH, but this allows you to switch them to 3B4H/3B5H and 3BAH.

2+3 This bit field selects the active clock. This also specifies the horizontal resolution of a pixel line because the dot clock rate is directly related to the number of pixels that can be displayed.

The dot clock rate of 0 MHz is reserved because it may only be used during a reset of the VGA card.

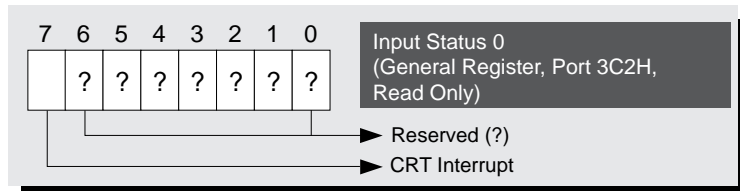
A reset using the sequencer's reset register must always be executed immediately after changing this register.

5 In the Odd/Even video modes (0, 1, 2, 3 and 7), this bit acts as the low bit for memory access. It determines whether odd or even addresses will be accessed in each bitplane. If this bit is 1 (default), then all bytes at even offset addresses are accessed. If it is 0, then odd addresses are accessed.

This bit loses its meaning if bit 1 in register 6 of the graphics controller sets chain mode or if bit 3 in register 4 of the sequencer controller sets chain4 mode.

6+7 These two bits specify the polarity between the horizontal and vertical rescan signals. This usually results in setting the specified vertical resolution.

Remember the VGA card's 200 pixel emulation mode cannot be set using this bit field because it's actually a 400 pixel mode that displays only half of the lines.



- 7 This bit indicates when a vertical rescan of the electron stream occurs in cases where this event triggers an interrupt. After a rescan, the value of this bit remains 1 until it is reset using bit 4 in the vertical rescan end register of the CRT controller.
- 1+2 These bits act as the light pen port on the EGA card. They aren't used on the VGA card. Bit 2 indicates the presence of a light pen and bit 1 indicates whether the button has been pressed. The position of the light pen can be read using registers 10H and 11H of the CRT controller.
- 3 This bit indicates the vertical rescan status, which allows a program to determine when certain register changes can safely be made.

CRT controller

The CRT controller is responsible for the picture displayed on your screen. It generates video signals for the monitor using the electron stream generated by the picture tube. It contains a number of registers that manage the timing of the electron stream's horizontal and vertical rescans.

Programmers usually aren't interested in these registers because the interactions between them are very complex. So, programming them should be left to the BIOS, which handles this automatically when you switch video modes. Registers, such as the offset register or the line compare register, are more useful to programmers. These can be used to create special video effects that aren't accessible using the BIOS. The following is an overview of the 25 CRT controller registers:

Number	Register Name	Number	Register Name
00H	Horizontal Total	0EH	Cursor Location High
01H	Horizontal Display End	0FH	Cursor Location Low
02H	Start Horizontal Blanking	10H	Start Vertical Rescan
03H	End Horizontal Blanking	11H	End Vertical Rescan
04H	Start Horizontal Rescan	10H	Light Pen Low (EGA only)
05H	End Horizontal Rescan	11H	Light Pen High (EGA only)
06H	Vertical Total	12H	Vertical Display End
07H	Overflow	13H	Offset
08H	Vertical Pel Panning	14H	Underline Location
09H	Maximum Scan Line	15H	Start Vertical Blank
0AH	Cursor Start	16H	End Vertical Blank
0BH	Cursor End	17H	Mode Control
0CH	Start Address High	18H	Line Compare
0DH	Start Address Low		

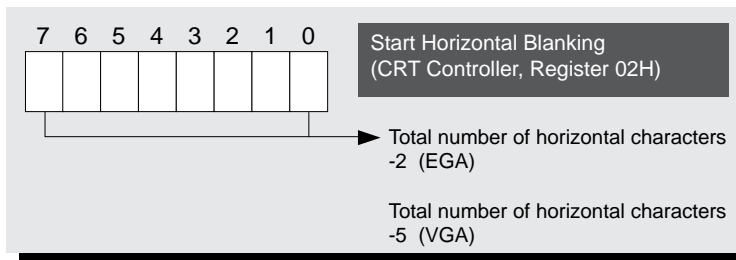
The CRT controller registers are addressed by an index register and a data register, which are located at port addresses 3D4H and 3D5H when the EGA or VGA card is in color mode. If the card is in monochrome mode, these registers are accessed at port addresses 3B4H and 3B5H.

As usual, the register number must be written to the index register prior to access. With a read access, the contents of the specified register can then be read in the data register. This doesn't apply to most of the registers on the EGA card, however. Only the two light pen registers can be read on this card; all other registers are write-only. All of these VGA registers can be read.

You can write to the index and data registers in a single 16-bit operation during a read access. The value for the index register must be in the low byte and the value for the data register must be in the high byte.

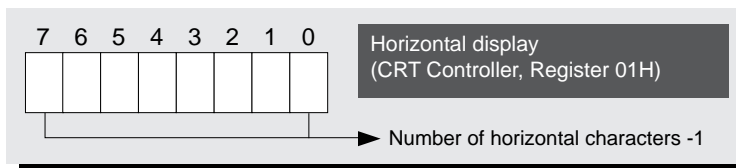
Once you've placed the register number in the index register, it remains valid for subsequent read or write operations. So, you don't have to enter the register number each time if you perform several consecutive operations on the same register.

Remember the first eight registers of the CRT controller on VGA cards can only be written if bit 7 in register 11H is set to 0. The BIOS will generally set this value to 1, prohibiting access to the first eight CRT registers.

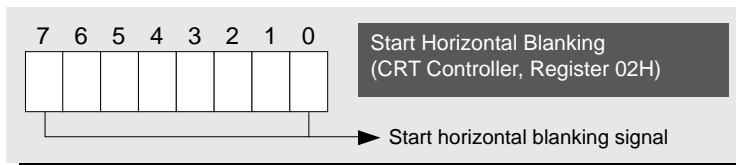


- 0-7 This specifies the total number of "characters" per screen line. The term "character" indicates an actual ASCII character in text mode. In graphics mode, "character" refers to a group of eight pixels. The total number of characters is calculated as the quotient of the bandwidth and the horizontal scan rate divided by the number of pixels per "character".

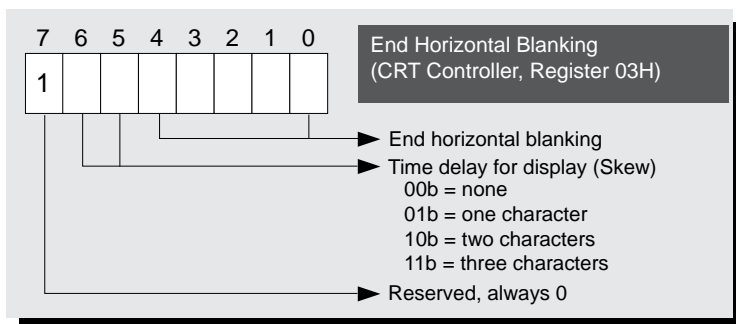
On an EGA card, the value in this register must be the number of total characters minus 2. For a VGA card, it's the total number minus 5.



- 0-7 This register specifies the actual number of horizontal "characters". The value in this register must be reduced by 1 with both EGA and VGA cards.



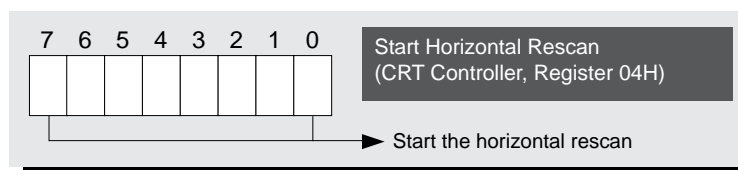
- 0-7 When the horizontal blanking signal starts, no more characters will be output because the electron stream from the picture tube has been shut off. Once again, the value is given in units of "characters." The first character at the left of the screen is assigned the number 0.



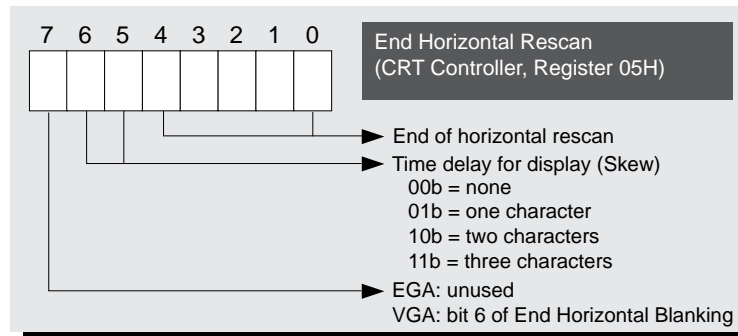
- 0-7 The end of the horizontal blanking is also described in terms of "characters." The first character output at the left of the screen is counted as 0. The end of the horizontal blanking always occurs before the start, so a maximum of six bits instead of eight are required. The sixth bit is in the end horizontal rescan register, which is number 05H in the CRT controller.

The value in this register is calculated as the sum of the values from the horizontal blanking register and the width of the horizontal blanking in "characters."

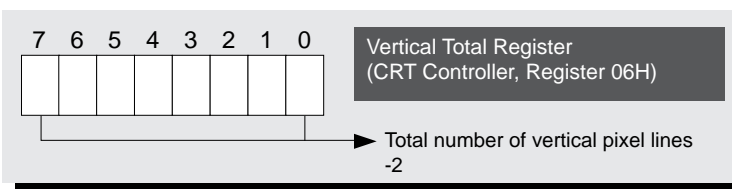
- 5+6 In certain instances, it may be necessary to build in a time delay (skew) to give the CRT controller enough time to read a character and its attributes from video RAM and then generate the corresponding pixel pattern with the character generator. The EGA card typically requires a skew of one character, but this usually isn't necessary with a VGA card.



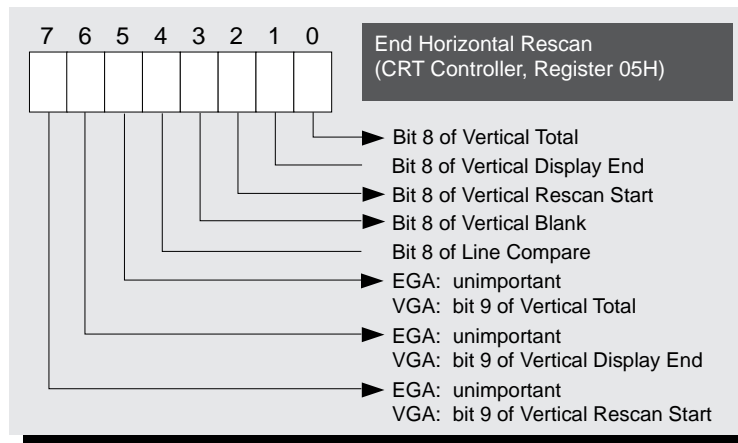
- 0-7 This register determines the "character" that triggers the rescan. This is used to center the picture horizontally.



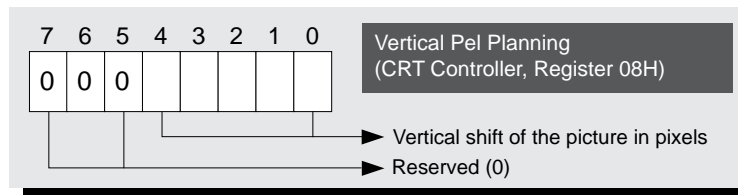
- 0-4 This specifies the number of characters where the end of the horizontal rescan is set. Since the end of the rescan is always before the start, only 5 bits are needed to code it. The units are again given in "characters".
- 5-6 A skew or time delay, can set for the end of the horizontal rescan just as for the end of the horizontal blanking. Since the skew will vary from card to card, you should never change these bits.
- 7 This bit is an expansion of bits 0 through 4 of the end horizontal blanking register. It's the highest bit of the group.



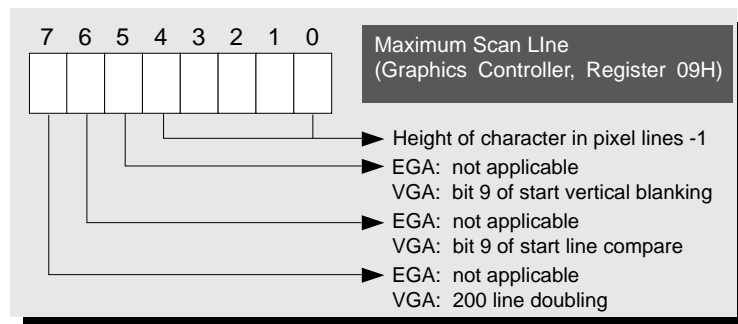
- 0-7 This register contains the number of pixel lines processed during a screen build. This includes those lines processed during the vertical rescan. For both EGA and VGA, the total number of pixel lines exceeds 256, so this register stores only the lower eight bits of the value. The ninth bit is found in the overflow register, which is number 07H. VGA cards also have a tenth bit, which is found in bit 5 of this same register. For both EGA and VGA cards, the value in this register must always be the actual total number of lines minus two.



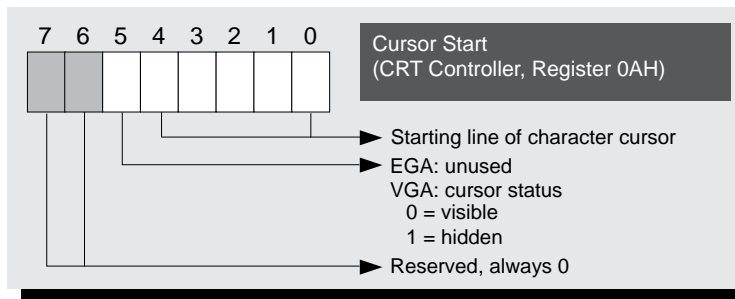
EGA and VGA cards need an overflow register because the number of vertical pixel lines exceeds 256 and therefore cannot be represented in a single eight bit register. The extra bit is kept in the overflow register for the registers that need nine bits. Additional overflow bits for the VGA card are stored in the maximum scan line register (index number 09H).



- 0-4 These bits are used to create smooth vertical scrolling by moving the entire picture up by a specified number of pixels. A value of 0 represents the normal picture location. Larger values indicate a corresponding upward shift.



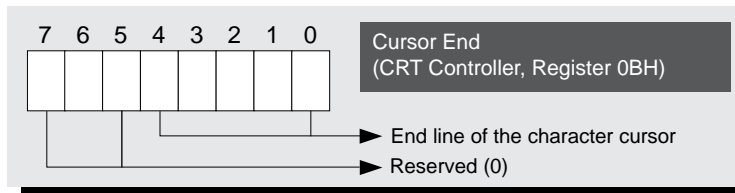
- 0-4 Determine the height of a character in text mode. The unit of measure is pixel lines. The value stored in this register must be the actual height minus 1. This bit field normally contains a value of 0 in graphics mode unless it's using a 200 line VGA mode. If so, the display of each line is doubled and this register contains a value of 1.
- 5 This bit isn't used on the EGA card. On the VGA card, it's used as bit 9 of the vertical blank start register. This bit also isn't used on the EGA card. For the VGA card, it is used as bit 9 of the line compare register.
- 7 The VGA card uses this bit for line doubling to obtain 400 lines in modes that are really 200 line modes. Unused on the EGA card. Do not change this bit when programming any registers involved in the vertical timing of the screen build.



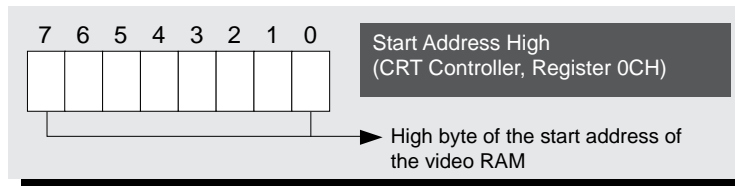
0-4 This specifies the starting line for the cursor; it begins with line 0. It can be a value from 0 to 31. If this value exceeds the actual character height, then the cursor will not be visible on screen.

If the starting line is greater than the end line (CRT register 0BH), then the EGA card will display a two-part cursor, but the VGA card won't display a cursor.

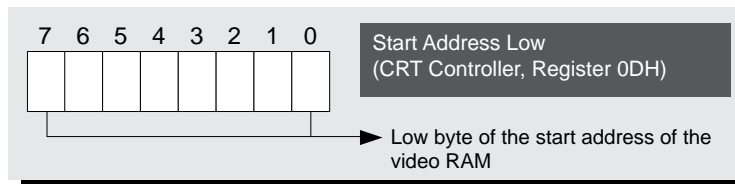
5 On the VGA card only, this bit can be used to explicitly hide the cursor. This bit isn't used on the EGA card.



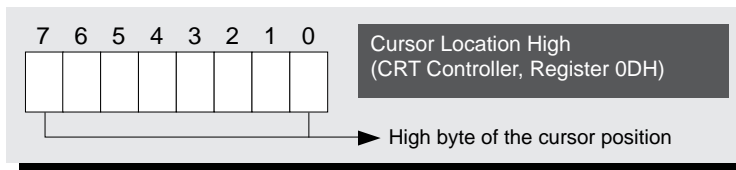
0-4 This bit field contains the last pixel line of the cursor. This value can also be from 0 to 31, but it may not exceed the character height. If the end line is less than the starting line (CRT register 0AH), then a two-part cursor will appear on an EGA card and a cursor won't appear on a VGA card.



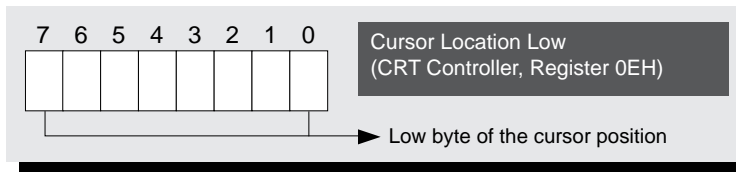
0-7 Together with register 0DH, this register represents the offset address at which the CRT controller will start to read the screen contents from the video RAM. This is the same as the start of the current screen page in video RAM. In odd/even mode, this value must be the actual offset value divided by two. In chain4 mode, it's the actual offset divided by four.



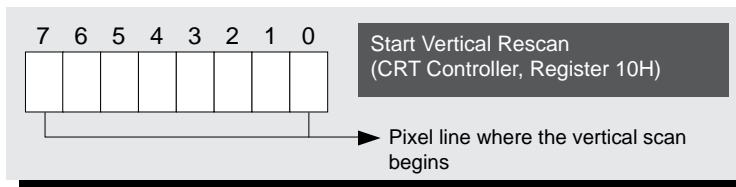
0-7 This register contains the low byte of the start address of the current screen page in video RAM. See register 0CH.



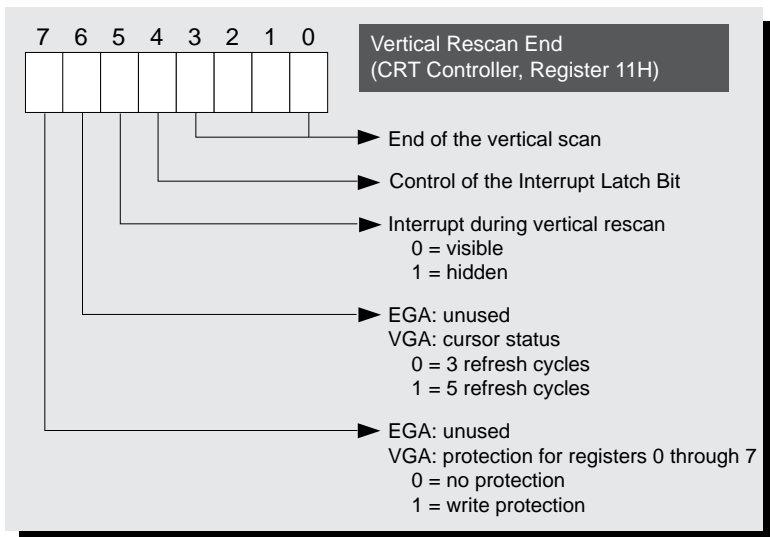
- 0-7 This register defines the current cursor position as an offset in the current screen page. The address given in this register must be the actual address divided by two. The high byte of the address is stored in this register and the low byte is stored in the next register.



- 0-7 This register contains the low byte of the cursor position in the current screen page. See also register 0DH.

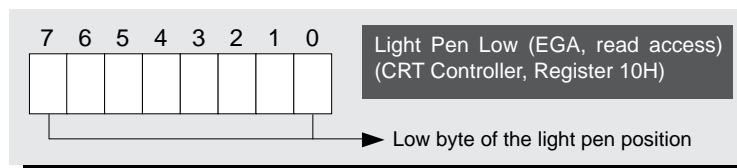


- 0-7 This contains the pixel line where the vertical rescan will begin. Since EGA and VGA cards manage over 256 lines, eight bits isn't enough room to store this number. The overflow register (register number 07H) contains a ninth bit and tenth bit (for VGA cards).



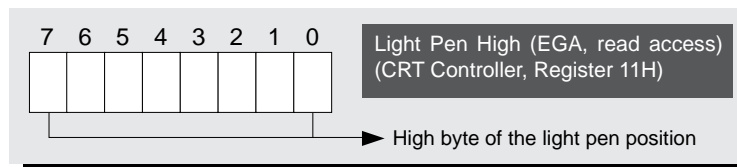
- 0-3 These bits contain the pixel line where the vertical rescan ends. The synchronization signal is switched off and a new screen build begins. Only four bits are used for this value, so the maximum line number is 15.

- 4 When a vertical interrupt is executed, bit 7 of the input status register is set to 1 to indicate the start of the vertical rescan. The interrupt bit remains active until it is reset by a value of 1 in this bit, which prohibits a new vertical interrupt.
- 5 This bit can be used to execute interrupt 2 with the start of every vertical rescan. However, you should remember that many VGA cards cannot generate a vertical interrupt.
- 6 Only VGA cards are able to change the refresh cycles per line from 3 to 5. Since this requires more time for each refresh of the video RAM, the VGA card operates with a smaller line frequency, which allows the use of monitors that cannot work with the normal VGA line frequency.
- 7 This bit locks and unlocks access to the first eight registers of the CRT controller on the VGA card. If this bit is set, these registers can be read but not written.

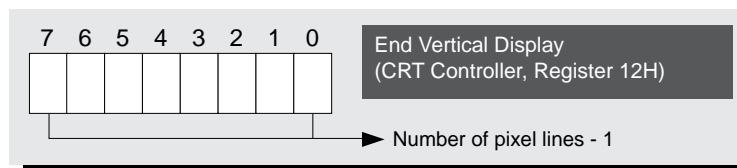


- 0-7 As we've already seen, the EGA card handles register 10H differently for read and write accesses. During a write access, it contains the start of the vertical rescan. With a read access, it returns the low byte of the current light pen position. As with the cursor position, this value represents the low byte of this address divided by two.

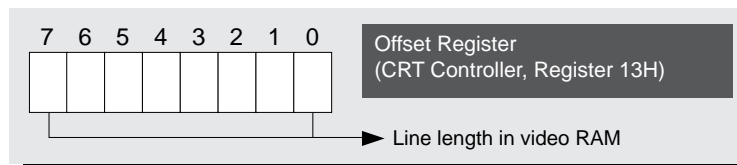
VGA cards, which don't support the use of a light pen, return the contents of the vertical rescan start register during a write access.



- 0-7 On EGA cards, this register returns the high byte of the light pen position with a read access. See also the description of register 10H.

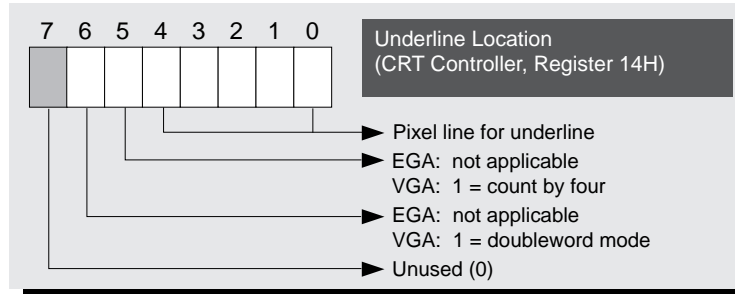


- 0-7 This register sets the number of the last pixel line in the screen build. This must be a nine bit number for EGA screens and a ten bit number for VGA. The additional bits required are found in the overflow register 07H.



- 0-7 The offset the CRT controller adds to the offset of the previous line at the start of each line is stored in this register. Depending on the address mode, this offset must be divisible by a certain factor. This factor is 2 in odd/even mode, 4 in chain4 mode and 1 in byte mode.

Normally, the value in this register corresponds to the length of a line in video RAM. In text mode, which is handled internally in odd/even mode, this value is 80 because a text line uses 160 bytes or 80 words. Larger or smaller values can also be given.



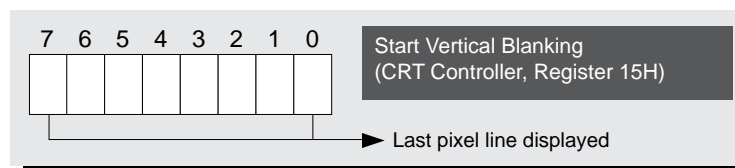
- 0-4 EGA and VGA cards working in monochrome modes can display characters on screen with underlining. This bit field indicates the pixel line in which the underlining will appear.

- 5 For VGA cards only, this bit determines whether the internal address counter will be incremented with every fourth tick of the character clock. If so, this bit must be set to 1 and bit 3 of the CRT mode register (count by two) must be set to 0. If the count by two bit is set to 1, then doubleword mode is ignored in any case.

- 6 This bit switches on the doubleword mode for VGA cards. It causes the access mode bit of the CRT mode register to be ignored.

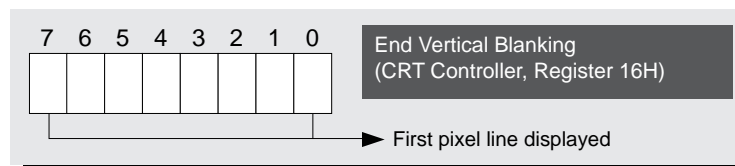
In doubleword mode, the address in the internal address counter is pushed up by two bits during the screen build. This moves bits 14 and 15 to bit positions 0 and 1. As long as the address counter is less than 4000H, all memory locations between 0000H and FFFCH where modulo 4 = 0 are addressed.

If the internal address counter reaches a value between 4000H and 7FFFH, then all memory locations between 0001H and FFFDH where modulo 4 = 1 are addressed. The same is true for the regions 8000H - BFFFH and C000H - FFFFH. The memory locations are then addressed where modulo 4 = 2 and 3, respectively.

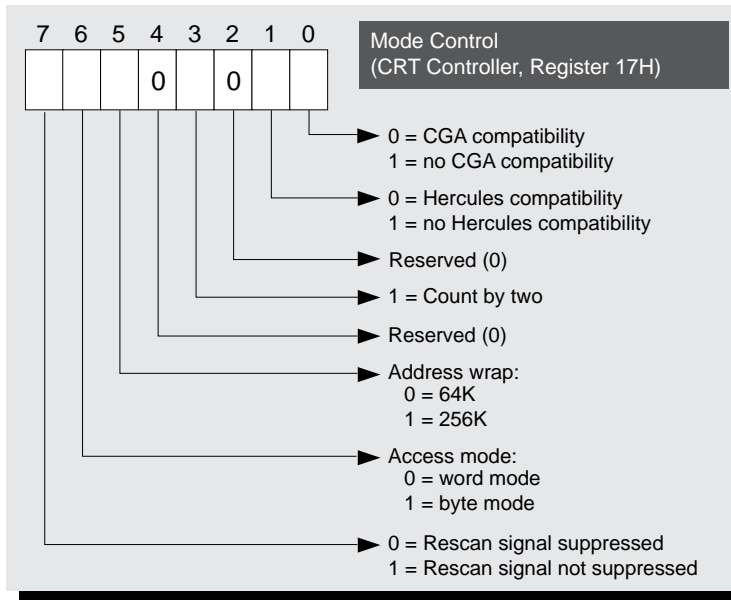


- 0-7 This register contains the number of the last pixel line plus 1. The ninth bit required for EGA and VGA cards is found in the overflow register (number 07H).

VGA cards require a tenth bit, which is found in the maximum scan line register (register number 09H).



- 0-7 This register contains the number of the first pixel line. When this line number is reached, the vertical blanking signal is switched off again so a new screen build can begin.



- 0 If a program sets this bit to 0, other registers can be set to emulate the CGA four color 320*200 pixel mode and CGA video RAM structure. The video RAM is divided into two blocks starting at offset addresses 0000H and 2000H. The first block contains the even-numbered lines and the second block contains the odd-numbered lines.

To emulate this process, setting this bit causes bit 0 of the internal row scan counter to be transferred to bit 13 of the internal address counter during the screen refresh. The counter contains the addresses in video RAM from which the CRT reads the screen information.

Bit 13 starts with a value of 2000H. So the CRT alternates between the address blocks starting at 0000H and 2000H, because bit 0 of the internal row scan counter alternates between 0 and 1. A requirement for this is the character height must be set to two by entering a value of 1 in the maximum scan line register.

- 1 This bit is an extension of bit 0. It must be set to 0 when emulating a foreign video mode that divides the video RAM into four blocks. This includes the Hercules graphics card and other CGA modes with 16 colors and 320*400 pixel resolution.

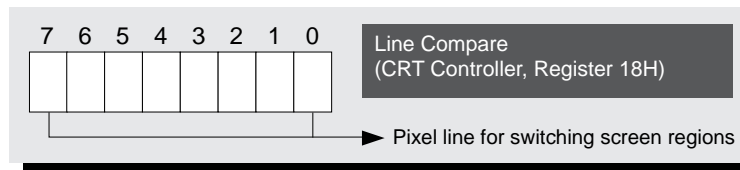
In these modes, the video RAM is divided into four blocks starting at offset addresses 0000H, 2000H, 4000H and 6000H. The block that contains a given line is determined by the modulo of the line number with 4. Line 0 is in the first block, the second line in the second block, the third line in the third block and the fourth line in the fourth block.

To emulate this structure, a procedure similar to that used with bit 0 copies bit 1 of the internal row scan counter to bit 14 of the internal address counter. Before doing this, you must be sure the character height has been set to four in the maximum scan line register. This ensures that bit 1 of the row scan counter will have a value of 1 so bit 14 in the address counter will be set to 1. The value in the address counter will be greater than 4000H in any case, so blocks two and three can be addressed.

- 3 If this bit is set to 0, then the internal address counter increments with each beat of the character clock. If assigned a value of 1, however, the counter only increments with every other beat of the character clock.

- 5 On EGA cards with only 64K of video RAM, this bit must be set to 0 to prevent overruns in word mode (see bit 6). Then bit 0 will be copied to bit 13, instead of bit 15 of the address bus.
- 6 Normally byte mode will be active and the value from the internal address counter is copied unchanged to the 16 address leads that determine which byte in video RAM will be addressed.

If this bit is set to 0, however, word mode is activated. In this case, the address bits from the internal address counter are moved one bit to the left and the highest bit is copied to the lowest address lead A0. As long as the address counter is less than 8000H, then the even bytes between 0000H and 0FFFEH are addressed. Larger values will address the odd bytes between 0001H and FFFFH.



- 0-7 This register can be used to divide the screen into two different regions in video RAM. The value stored in this register represents the line number where the first region ends and the second begins. When this line is reached, the CRT controller sets the internal offset address for querying screen information from video RAM back to 0.

When a new screen refresh begins, the screen region with the addresses indicated in CRT registers 0CH and 0DH is displayed.

Since both EGA and VGA cards are able to display more than 256 lines on screen, eight bits isn't sufficient to represent the line number. A ninth bit is stored in the overflow register (07). A tenth bit is stored for VGA cards in the maximum scan line register (09).

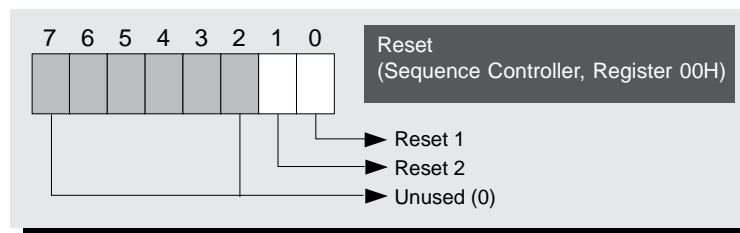
Sequencer controller

The registers of the sequencer controller are accessed in the usual way using a data register and an index register. The index register is located at port address 3C4H. This is immediately followed by the data register, which is located at port address 3C5H.

Number	Register Name
00H	Reset
01H	Clocking Mode
02H	Map Mask
03H	Character Map Select
04H	Memory Mode

As opposed to the other EGA and VGA controllers, the sequencer controller handles a number of miscellaneous tasks and cannot be easily categorized. Its responsibilities range from video RAM memory access and the management of bitplanes to selecting the currently active character table. It's also responsible for refreshing the video RAM.

The sequencer controller on EGA and VGA cards has five different registers, as shown in the table on the left. Remember, the contents of the register can only be read with VGA cards. This isn't possible with an EGA card.

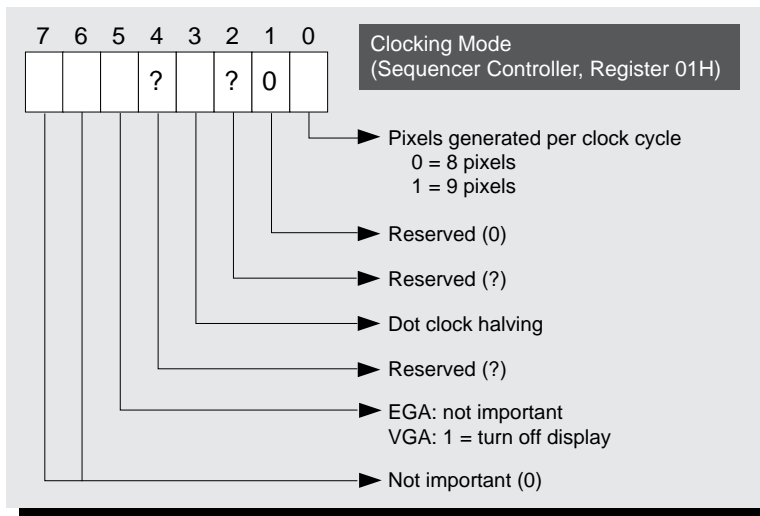


- 0 This bit usually contains a value of 1. It can be set to 0 to execute a sequencer controller reset, which ends its activity. This ends the generation of horizontal and vertical synchronization signals, causing the screen to go

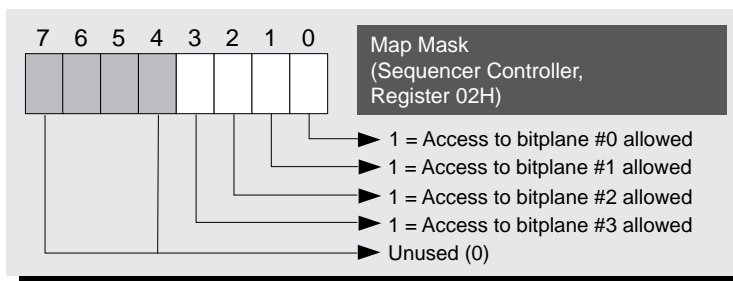
black. Also, the character map select register is set to 0 and the video RAM refresh is switched off. To avoid losing the contents of the video RAM, this bit should be set back to 1 after a maximum of 20 or 30 microseconds. This brings the sequencer controller back to life.

A reset of the sequencer controller is required prior to programming bits 0 and 3 of the clocking mode register of the sequencer controller, as well as bits 2 and 3 of the miscellaneous output register.

- 1 This bit is also used to reset the sequencer controller, except that it doesn't also reset the character map select register. So, you can use this bit instead bit 0 to reset the sequencer controller, but both bits 0 and 1 must be set back to 1 for the sequencer controller to be able to continue its work.

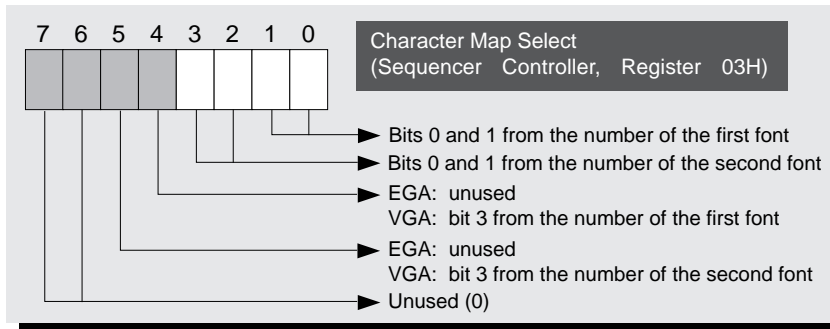


- 0 This bit sets the number of horizontal pixels that are generated by the CRT controller during each clock cycle. In the graphics modes and with the color text modes of the EGA card, this is always eight. For VGA text modes and for the use of an MDA monitor with an EGA card, this bit must be set to 1 to generate nine pixels per clock cycle.
- 3 Setting this bit to 1 results in halving the dot clock rate. This happens automatically when the BIOS is used to switch on the 320*200 pixel mode for CGA emulation. This bit will be 0 for all other modes (including the 320*200 256 color mode).
- 5 On a VGA card, setting this bit to 1 will switch off the video signal. This makes the screen go black and allows the CPU unlimited access to the video RAM.



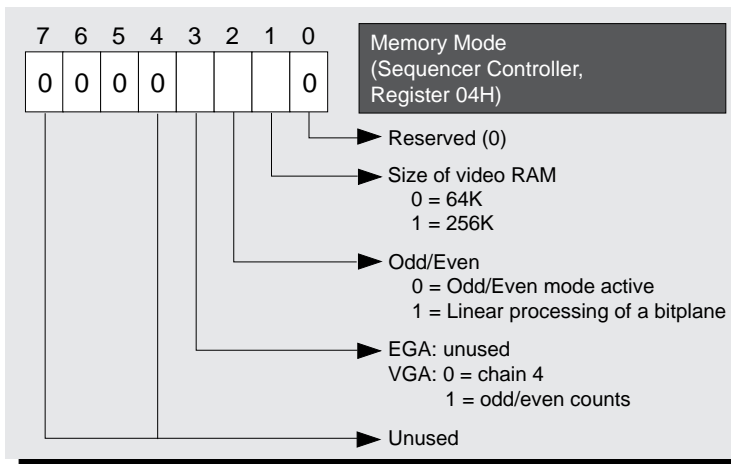
- 0-3 Each of these bits either blocks or enables access to a bitplane. This is important for accessing the video RAM with the various read and write modes. During a read access, the status of this bit determines whether a byte in

the bitplane should be filled with the contents of the corresponding latch register. Conversely, during a read access, this bit determines whether the byte in question should be copied from the bitplane to the latch register or if the contents of the latch register should remain unchanged.



0+1+4 These bits determine the number of the character table that will be used for all characters that have a value of 0 for bit 3 in their attribute byte. Bit 4 isn't used on EGA cards, so font selection is limited to numbers 0 - 3. Bit 4 is used on VGA cards to allow the selection of font numbers from 0 - 7.

2+3+5 These bits determine the number of the character table that will be used for all characters where bit 3 in the attribute byte is set to 1. Again, the third bit (bit 5) is only needed for VGA cards.



1 This bit applies only to EGA cards because all VGA cards come with 256K of video RAM. You can also safely assume that few, if any, EGA cards with only 64K are still being used.

2 This bit is used to set the division of odd and even memory addresses into different bitplanes. This is known as odd/even mode. In this mode, access to even addresses in video RAM are automatically routed to bitplanes 0 and 2. Access to odd addresses go to bitplanes 1 and 3. In both cases, the low bit of the offset address will be expanded by the page bit (bit 5 in the miscellaneous output register) before this occurs. As always, access to the various bitplanes can be suppressed using the map select register of the sequencer controller.

If this bit contains a value of 1, the memory addresses aren't divided into odd and even groups and the bitplanes can be processed in a linear fashion. Remember the contents of this bit should always correspond with the odd/even bit in register 5 of the graphics controller.

- 3 The VGA card uses an expansion of the odd/even mode known as chain4 mode. This is a sort of doubled odd/even mode. It's used mostly in 256 color graphics modes to create a linear video RAM starting at A000H. This is from a program's point of view only and in reality the video RAM is still divided into four bitplanes. This only pertains to access to the video RAM using the CPU. It has no effect on video RAM access through the CRT controller.

As with odd/even mode, access to the video RAM is based on the addresses of memory locations within one of the four bitplanes. The two low bits of the offset address are masked (set to 0) so only locations within the bitplanes that are divisible by four can be accessed. The number of the bitplane to be accessed is determined by the value of two bits that were masked, which is the same as module 4 of the offset address.

A bitplane can be accessed only if it's freed using the map mask register of the sequencer controller. If chain4 mode is active, the bits used to control the odd/even mode are ignored. They become valid again when chain4 mode is switched off.

Attribute controller

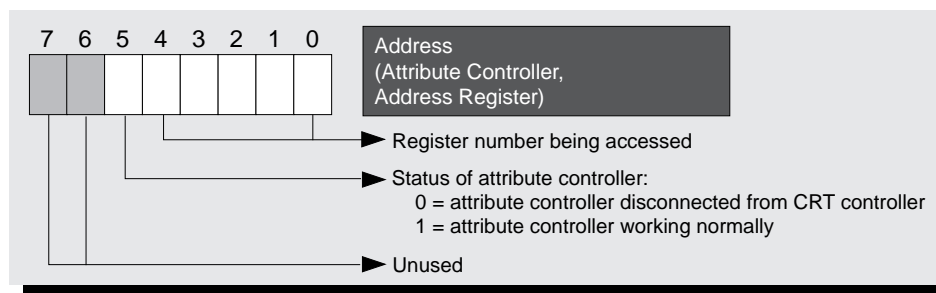
The attribute controller prepares the color signals for the screen. It manages the palette registers and the other registers needed for generating color signals. The registers of the attribute controller are accessed using a combined address and data register. The register located at port address 3C0H is used for write operations. For read operations, which are only possible on the VGA card, the register at port address 3C1H is used. To access an attribute register, first the number of the desired register must be written to port 3C0H or 3C1H. With a read operation, the result can then be read directly from port 3C1H. For a write operation, the new contents of the register must be sent using port 3C0H.

Number	Register Name
00H-0FH	Palette Register
10H	Mode Control
11H	Overscan Color
12H	Color Plane Enable
13H	Horizontal Pel Penning
14H	Color Select (VGA only)

For read access to a register, the combined index and data register at port address 3C1H requires only the register number. The register for write operations requires an additional bit to determine the status of the attribute controller. If this bit is set to 0, then the connection between the attribute controller and the CRT controller is broken and the screen is black or whatever color is contained in the overscan register of the attribute controller. Separate color signals for the characters or pixels on screen will no longer be produced.

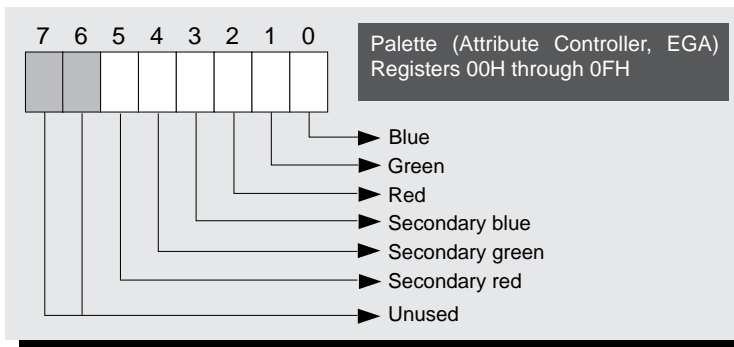
The attribute controller on an EGA card has 20 different registers and the VGA card has one additional register. The assignments for some registers differ between EGA and VGA cards because VGA cards require using the DAC

color table to generate color signals. EGA cards don't have the DAC color table. Like the sequencer controller, the registers can only be read on a VGA card. These same registers on an EGA card are write-only.

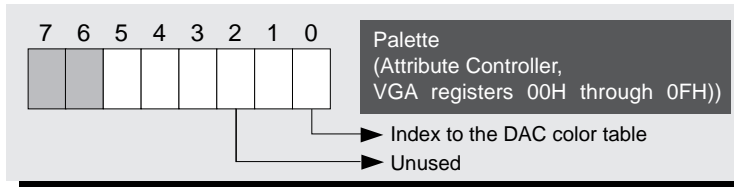


- 0-4 These bits address the various registers of the attribute controller. This is done in the same way as with other EGA and VGA controllers.
- 5 If this bit contains 1, the attribute controller works normally. If it contains 0, then the attribute controller is disconnected from the CRT controller, which causes the screen to go black or to become the color stored in the overscan register.

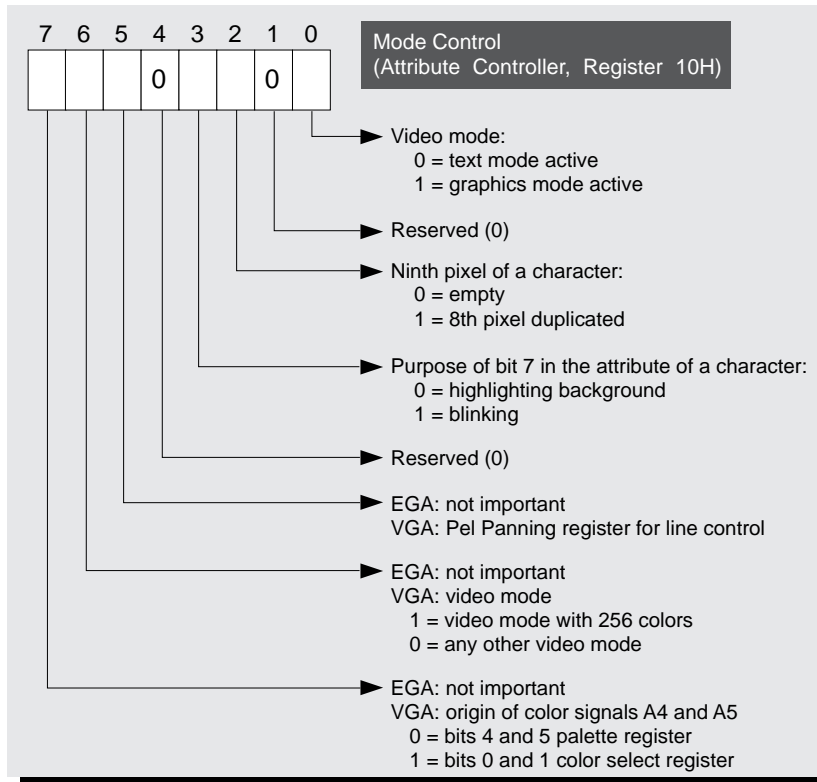
This is the only time when the palette registers can be accessed directly by the CPU to be read or written.



0-5 On an EGA card, the 16 palette registers contain actual 6-bit color codes. This allows for 64 different colors.

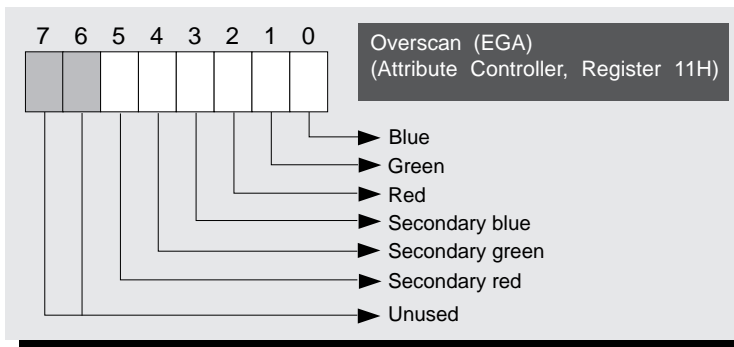


0-5 On a VGA card, a palette register stores an index to the DAC color table instead of an actual color code. The attribute controller uses this index value to load the color assigned to a character or pixel from the DAC color table and send the appropriate color signal to the monitor.

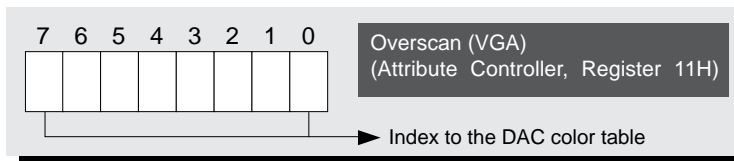


- 0 This bit is used to tell the attribute controller whether a text mode or a graphics mode is active.
- 2 This bit is only meaningful for text modes that use a nine pixel character width. This bit determines the meaning of the ninth pixel. The character tables in ROM only store eight pixels per line for each character.

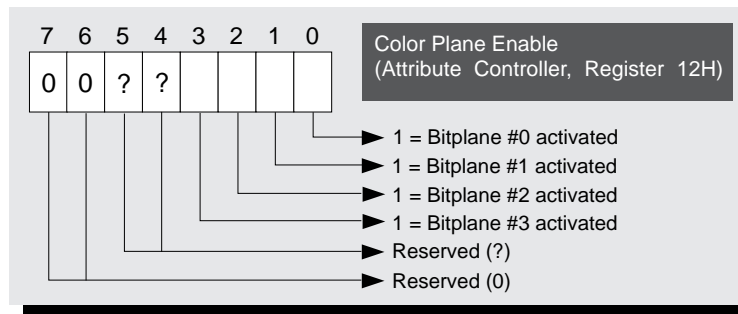
If this bit is 0, then the ninth pixel remains empty. This creates a space of one pixel between two subsequent characters. If this bit is set to 1, then the contents of this pixel is determined on a character by character basis. If the character is a special border or frame character (ASCII codes between C0H and DFH), then the contents of the eighth pixel are copied to the ninth. The ninth pixel remains empty with all other characters.
- 3 When this bit is set to 1, a cursor will appear in text mode. The cursor will remain visible for 16 screen generations and then invisible for the next 16. Depending on the frequency of the screen regenerations, this results in a cursor that blinks every 1/3 to 1/4 second.
- 5 If the line register has been used to split the screen into two halves, this bit determines whether the horizontal and vertical panning should pertain to only the first screen region or to both. Both screen regions are affected with a value of 0. This applies only to VGA.
- 6 When this bit is set to 1, the attribute controller is set for a 256 color graphics mode. The color information is no longer read directly from the palette register, but rather with an index to the DAC color table.
- 7 In color modes with 16 or fewer colors, this bit determines whether the color signals A4 and A5 for addressing the DAC color register should be taken as signals A0 through A3 from the palette register or as constants from bits 0 and 1 of the color select register of the attribute controller.



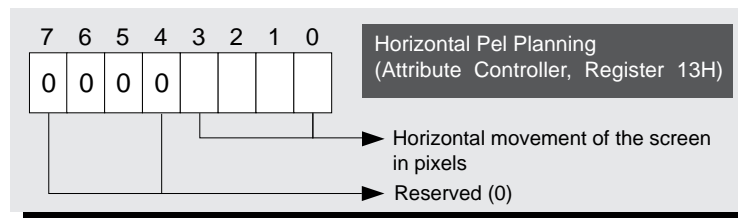
- 0-5 On EGA cards, the color for the screen border is stored here in the same format used with the palette registers.



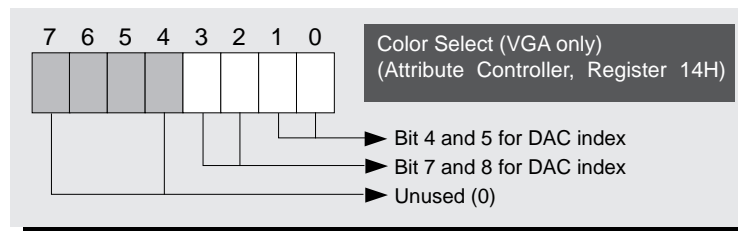
- 0-7 On a VGA card, the frame color is selected from the DAC color table. Any of the 256 colors in the table can be accessed using this register.



- 0-3 The four bits in this bit field turn each bitplane on and off for the transfer of color information from the video RAM to the attribute controller. This can be used together with programming the palette or DAC color registers to exclude certain pixels from being displayed.



- 0-3 This is where the counter for horizontal pel panning is stored. This value is used when moving the visible screen display to the left.



- 0+1 These two bits are used as bits 4 and 5 of the DAC index number when bit 7 in the mode control register of the attribute controller is set. They then replace bits 4 and 5 from the corresponding palette register.
- 2+3 These two bits are bits 6 and 7 of the DAC color table index in all graphics modes with less than 256 colors, regardless of the contents of other registers or bit fields.

Graphics controller

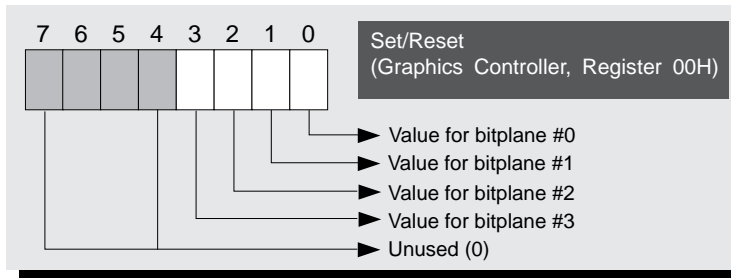
The graphics controller is used in all read and write accesses between the CPU and the video RAM using the latch registers. The registers of this controller therefore determine the current read/write mode and store the various parameters for each mode.

The nine registers of the graphics controller are accessed using a data register and an index register. They're found at port addresses 3CFH and 3CEH. As usual, the number of the desired register must first be loaded in the index register (3CEH). This determines the index

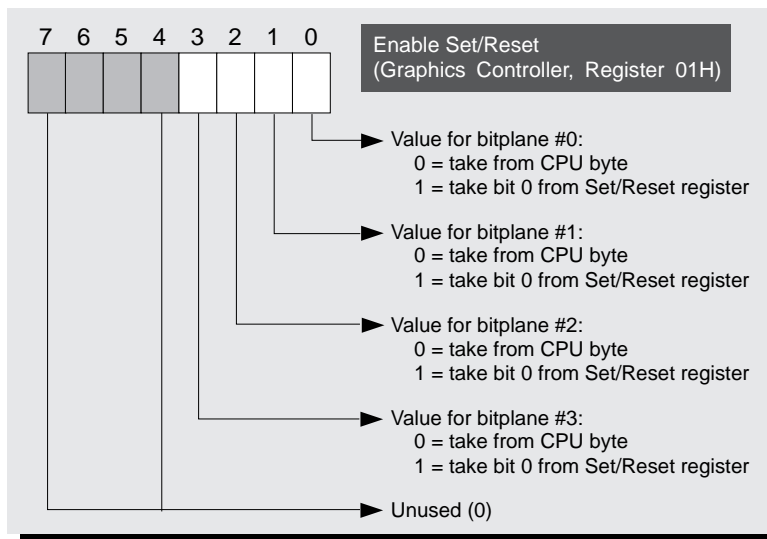
Number	Register name	Number	Register name
00H	Set/Reset	05H	Graphics Mode
01H	Enable Set/Reset	06H	Miscellaneous
02H	Color Compare	07H	Color Don't Care
03H	Function Select	08H	Bit Mask
04H	Read Map Select		

that will then be read or written using the data register (3CFH).

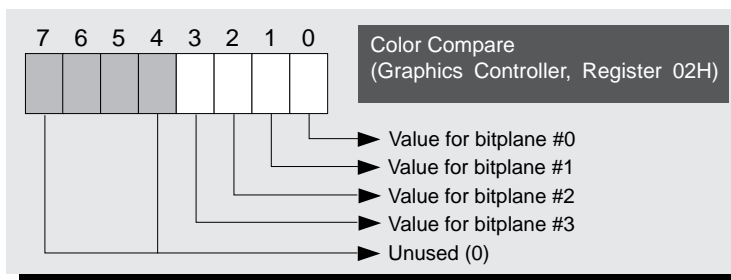
As with the registers of the other controllers, read access is only possible with VGA cards. EGA graphics controller registers are write-only. The following is an overview of the graphics controller registers:



- 0-3 These bits are used for video RAM access in write mode 0. They determine the value to be written to bit 8 in each bitplane if the source for this bit has been selected as the set/reset register. The source for bit 8 is determined by the enable set/reset register (number 02H).

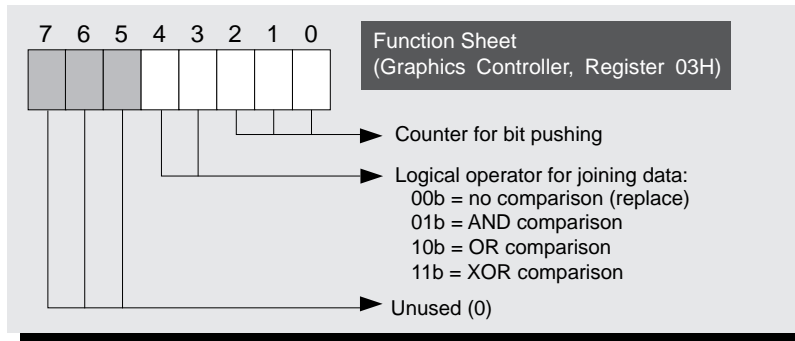


- 0-3 When working with write mode 0, these bits determine whether the value of bit 8 in the bitplane being accessed should be taken from the CPU byte or from the set-/reset register (number 00H).



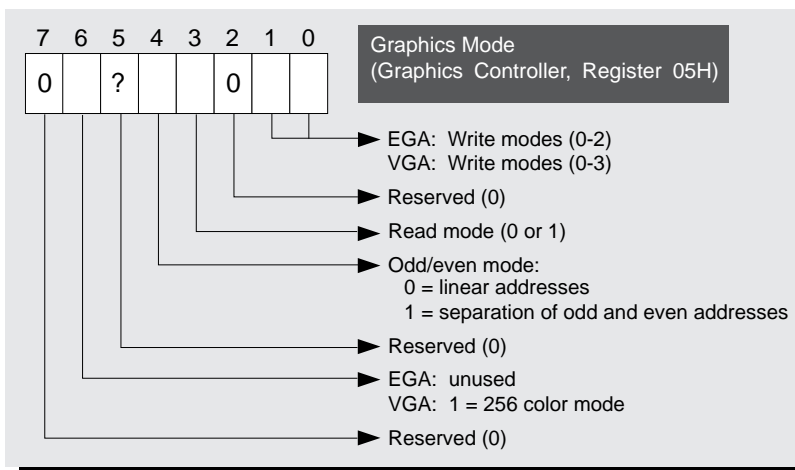
- 0-3 These bits are important for read access to the video RAM in read mode 1. In this mode, the four bytes read from the four video RAM bitplanes are organized into eight groups of four bits, each of which represents the color of

one pixel. The eight resulting color codes are compared individually with the contents of this register and the result of this comparison is sent to the CPU.

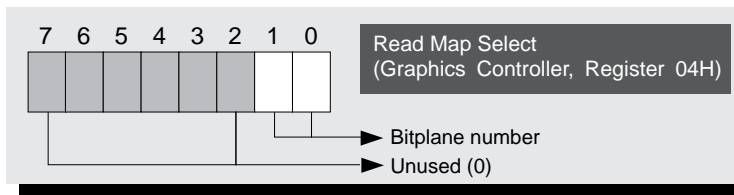


- 0-2 For write access to the video RAM in write modes 0 and 3, this bit field determines the number of bit positions the CPU byte must be pushed to the left before it can be joined with the contents of the latch register.
- 3+4 In write modes 0 and 2, this bit field determines the logical operator used to join a CPU byte and the data from one of the four latch registers before it's written to one of the bitplanes.

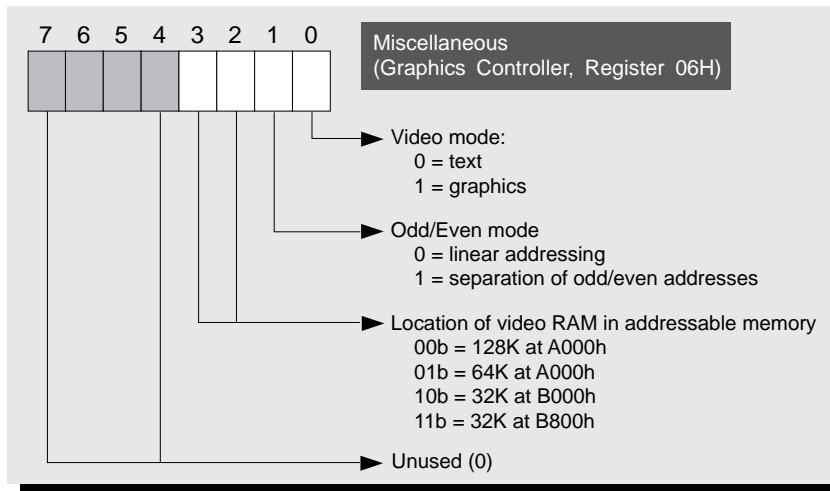
The four bits in the bit mask register which correspond to the four latch registers determine whether the CPU byte and latch register will be combined. If the corresponding bit from the bit mask register has a value of 1, then the logical operator indicated by this bit field is used. Otherwise, the values are not combined.



- 0+1 These bits store the number of the write mode used for write access to the video RAM, regardless of the current video mode.
- 2 This bit stores the number of the currently active video mode.
- 3 This bit tells the graphics controller whether an odd/even mode is active or if the bitplanes are to be addressed in a linear fashion. This bit must be loaded with the same value found in bit 2 of the memory mode register of the sequencer controller.
- 5 For VGA, this bit indicates whether a 256 color mode is active. If it is, the graphics controller must use a different method for passing color information on to the attribute controller.



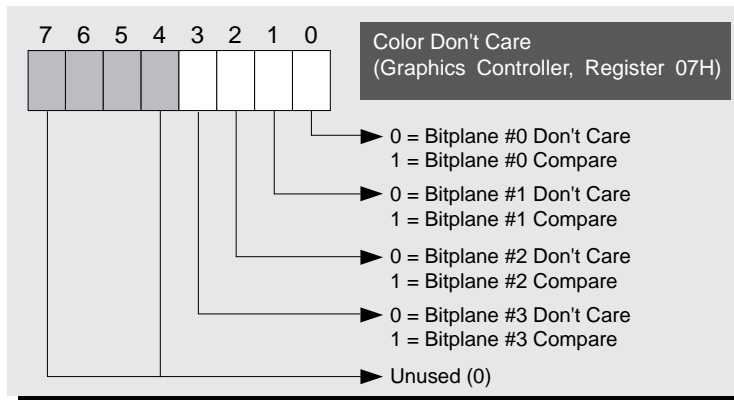
0+1 For read access in read mode 0, these two bits store the number of the latch register (which is the same as the number of the associated bitplane) that will be copied to the CPU. Read accesses in read mode 1 and chain4 mode are not affected.



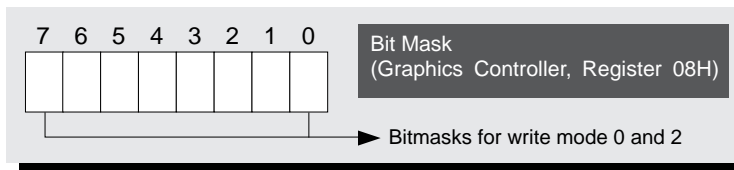
0 This bit indicates whether a text or graphics mode is currently active. This is so the graphics controller will know whether to use the internal latch registers for converting ASCII codes into pixel patterns for text modes.

1 This bit also indicates whether an odd/even mode is active or if linear addressing of the bitplanes is being used.

2+3 This bit field determines where the video RAM will be located in the processor's addressable memory. The BIOS sets this value correctly depending on the video mode and the type of monitor being used.



0-3 These bits are important for read accesses to the video RAM in read mode 1. They indicate which bitplanes will be included in the color comparison.



- 0-7 For write access to the video RAM in write modes 0 and 2, these bits determine which bits from a given bitplane should be written unchanged to the corresponding latch register and which must first be combined with other data (from the CPU or the set/reset register) using the function select register. This happens in all four latch registers to those bits for which the corresponding bit in this register is set to 1.

Digital to Analog Converter (DAC)

The DAC is found only on VGA cards. Its job is to convert the digital color values into analog color signals for the monitor. The DAC is the last step in color generation for VGA. Each palette register contains an index to one of the 256 color registers in the DAC.

A DAC color register consists of 18 bits: Six bits for each of the basic colors red, green and blue (RGB). This allows for 262,144 different colors. The color registers of the DAC are accessed using the registers listed in the following table. Unlike other EGA and VGA controllers, a single port address is used instead of separate index and data registers.

Number	Register Name
3C8H	Pel Write Address
3C7H	Pel Read Address
3C7H	DAC State
3C9H	Pel Data
3C6H	Pel Mask

To write to one of the DAC color registers, first you must enter its number (0 - 255) in the pel write address register. Then send the data for the new color to the pel data register. The pel data register is only eight bits wide and cannot receive all 18 bits needed to define a color at once. Therefore, the six bits of the red color component are sent first, followed by the six green bits and finally the six blue bits.

Within a program, usually all 256 DAC color registers (or at least a good number of them), instead of only one or two, are loaded. So, the value in the pel write address register is automatically incremented after the data for the three color components has been sent to the pel data register. This allows you to proceed directly with the

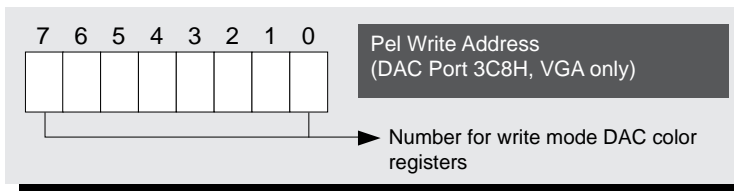
color components for the next DAC color register without having to access the pel write address register again.

The automatic incrementing of the DAC color register number stops when you enter a new register number in the pel write address register. This starts a new load operation. The number of the DAC color register that is currently being processed can be queried in the pel address register.

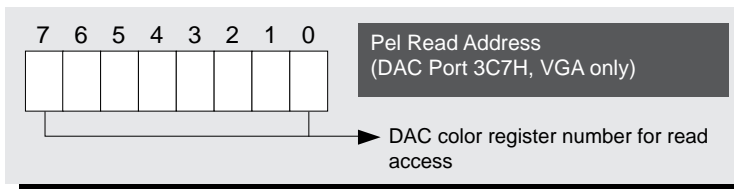
This procedure is used for both write and read access to the DAC color registers. For reading, simply load the pel read address register with the number of the first DAC color register that you want to read. The red, green and blue components of each DAC color register can then be read from the pel data register. There must be a slight pause between the three read accesses to the pel data register. In an assembler program, this can be done with a jump to the next command.

As we've described, the contents of the pel read address register are also incremented after the contents of the current DAC color register have been read. The attribute controller has no access to the DAC color registers while they are being read or written. So, the display must be switched off during this time so a blizzard doesn't appear on screen.

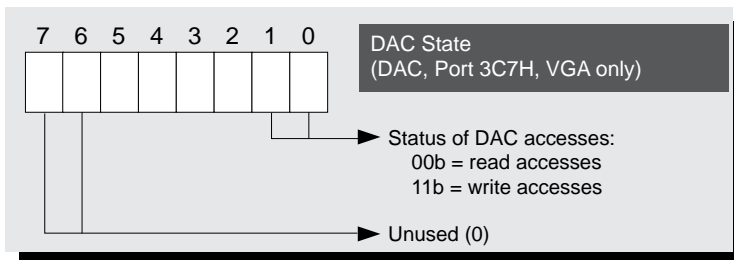
The following pages describes each DAC register:



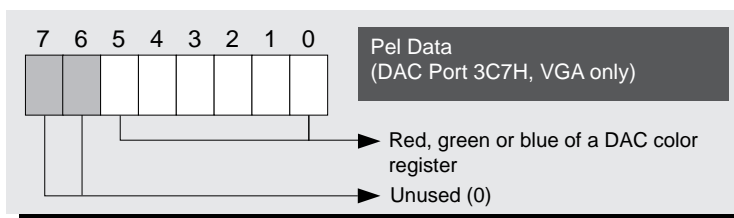
- 0-7 This register stores the number of the DAC color register that is to be written. The value from the pel data register is copied to this DAC color register in the next step of the write operation. This value is automatically incremented so all (or a large portion) of the DAC color registers can be loaded without having to access this register again.



- 0-7 This register stores the number of the DAC color register that is to be read. The color component values from the DAC color register can be read from the pel data register in the next step of the read operation. This value is also automatically incremented.

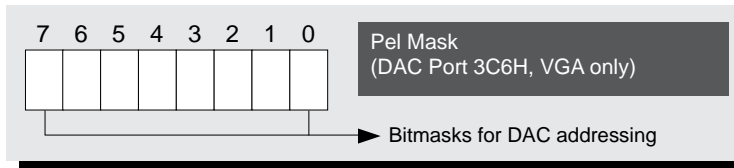


- 1+2 These bits indicate whether the DAC is currently in read or write mode.



During a write operation on one of the DAC color registers, this register must be loaded with the three color components in sequence: first red, then green and finally blue. Each color component requires only six bits, so it is the lower six bits that are actually copied to the DAC color register.

During a read access to a DAC color register, this register will first have the six bits of the red, then the green and finally the blue color components available for querying.



- 0-7 If the attribute controller wants to read the contents of a DAC color register while generating colors during a screen refresh, the contents of this register must be combined with the number of the desired DAC color register using a logical AND operator. When animating, for example, this allows you to create entire color groups using certain DAC color registers at a low level.

Normally, this register contains the value FFH, so accesses to the DAC color registers aren't changed.

Super VGA Cards

Although it took more than a year for chip manufacturers to unlock the secrets of the IBM VLSI chips to create EGA-compatible video cards, it was only a few months before the first VGA-compatible cards arrived on the market. Just as with EGA cards, a battle for the market share of VGA-compatible video cards began. Prices decreased and manufacturers were pressured to come up with new features that set their card apart from the others. This is how the Super VGA cards were developed. Although these cards are compatible with the IBM VGA standard, they use many hardware registers that aren't defined in this standard to provide extra features and performance. Some of the enhanced capabilities of Super VGA are:

- More colors in the normal VGA graphics modes
- Higher resolution graphics modes
- Text modes with more columns and lines
- Hardware cursors in graphics modes
- Hardware zooming displayed on the screen in graphics modes

Unfortunately, the manufacturers of these cards didn't develop standards for these advanced features. With a standard, these features could be accessed from the ROM-BIOS. As developers of the expanded capabilities of Super VGA cards know, one program won't work the same way with two different cards. So, to ensure that a program will run with all Super VGA cards, separate routines, which are designed to meet each card's requirements, must be created.

In this section we'll discuss some important aspects of Super VGA cards. Instead of discussing the many cards that are available, we'll end this chapter talking about the VESA Super VGA Standard. This standard was created in 1990 by the major manufacturers of Super VGA cards in an attempt to create a practical tool for Super VGA programming. This standard enables your program to work with Super VGA cards from several different manufacturers.

Although hundreds of companies make Super VGA cards, only about twelve suppliers provide the chips needed for building Super VGA cards. The most important suppliers include the following:

- ATI
ATI manufactures the VGAWONDER which emulates all earlier graphics standards (including Hercules).
- Chips & Technologies
This company manufactures the NEAT chip set.
- Genoa
This is the one company that actually uses the name "SuperVGA" on its Super VGA line of cards.
- Headland
(also known as VideoSeven).

- Tseng
Probably the leading manufacturer or supplier, Tseng supplies the ET30000 and ET4000 VGA chips which are installed in several VGA cards.
- Paradise
A division of Western Digital, a manufacturer of hard drives and other mass storage systems.

Although there are only six major manufacturers of VGA and Super VGA chip sets, the differences between Super VGA cards aren't limited to the use of six different chip sets. Nearly every chip set allows a card manufacturer some freedom in mode selection and addressing BIOS functions.

Super VGA text modes

Of all the various Super VGA text modes, the 25 line by 132 column mode that's supported by many cards is one of the most interesting. There are also other text modes that have 132 columns and higher numbers of lines. Several 80 and 100 column modes, with varying numbers of lines, are also supported. The highest resolution text mode supports 160 columns and 50 lines. Whether you can read any of the characters on the screen is another matter.

Expanded text modes of various Super VGA cards			
Columns	Rows	Columns	Rows
80	30	132	25
80	34	132	28
80	43	132	30
80	60	132	43
100	37	132	44
100	43	132	50
100	60	160	50
100	75		

The table on the left shows a sampling of some of the text modes available on various Super VGA cards.

These text modes are usually initialized with function 00H of the BIOS video interrupt. However, different cards may have different code numbers for each mode. This is unfortunate because the video RAM in these modes is structured as it is in the normal text modes. This would make it possible to use the expanded modes simply by setting a wider line length in video RAM.

If you want your DOS programs to have flexible screen resolution, you should use the functions of the VESA BIOS. Or, you can let the user decide. Some configuration programs allow the user to set the screen resolution at the DOS level.

Then the user can simply pass the screen resolution, as a parameter, to the program when starting it from DOS or enter the value in a configuration file the program reads after it starts.

Super VGA graphics modes

Super VGA graphic modes also offer a wide variety. The standard VGA modes are supported with more colors, for example 256 instead of only 16. However, there are new graphics modes with higher resolutions, up to 1024x768 pixels.

Unfortunately these higher resolutions are slower. For example, a 640x480 pixel mode uses about 300,000 pixels and an 800x480 mode uses almost 500,000. The processor still must process each pixel, which prevents it from performing other tasks. This is why many Windows users prefer the old standard VGA 640x480 pixel mode. The Super VGA modes simply take too long to paint the screen.

In addition to modes with resolutions like 720x396 or 960x720, the same three modes are included on almost all Super VGA cards:

- 640x400 pixels with 256 colors
- 800x600 pixels with 16 or 256 colors
- 1024x768 pixels with 16 colors

The following table shows these modes represent only a portion of the various Super VGA graphics modes available on different cards:

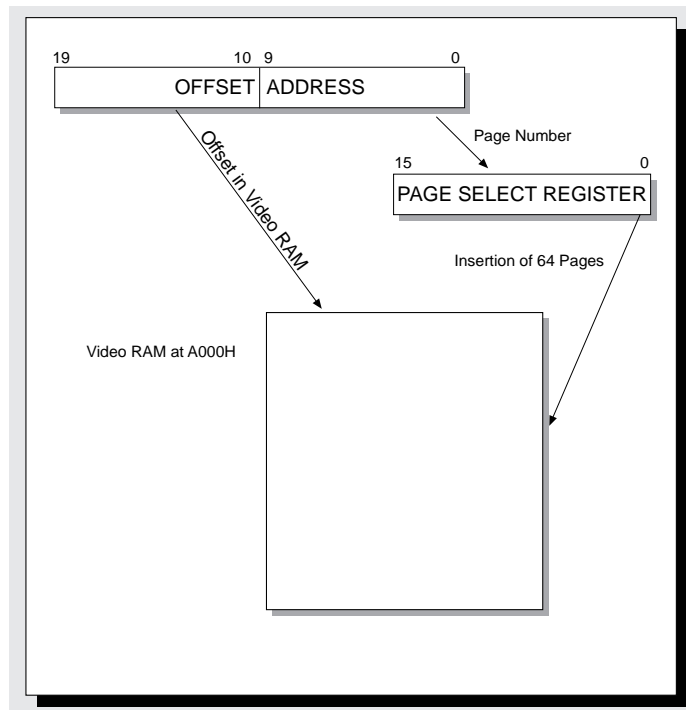
Expanded graphic modes of various Super VGA cards							
Resolution	Colors	Pixels	Memory	Resolution	Colors	Pixels	Memory
512x480	256	245,760	256K	720x540	256	388,800	512K
640x400	256	256,000	256K	752x410	16	308,320	256K
640x480	256	307,200	512K	800x600	16	480,000	256K
720x396	16	285,120	256K	800x600	256	480,000	512K
720x512	16	368,640	256K	960x720	16	691,200	512K
720x512	256	368,640	512K	1024x768	16	786,432	512K
720x540	16	388,800	256K	1024x768	256	786,432	1 Meg

Regardless of which graphics modes a Super VGA card supports, a given mode can be used only if the proper monitor and sufficient video RAM are available. A normal fixed-frequency VGA monitor is sufficient for the 640x480 modes, but the higher resolution modes require a multisync monitor. When you reach resolutions as high as 1024x768, you'll need an XL monitor. This is a special kind of multisync monitor; that's often used with CAD/CAM applications. A resolution of 1024x768 would represent the lower end of its capabilities.

Structure of the video RAM in Super VGA modes

When you look at all the Super VGA graphics modes, they can be divided into modes that use either 16 or 256 colors, just like the standard VGA graphics modes. In both cases, the structure of the video RAM is the same as in the related standard VGA modes. This means the color of a pixel is determined by four bits from the four bitplanes in the 16 color modes. The 256 color modes use an entire byte to define the color of a pixel. The address of a pixel is the result of multiplying the line number by the line length and adding the column number. The only difference between these modes and the standard VGA modes is the length of each line in video RAM. This makes each screen page longer, which results in a higher resolution.

The 64K limit at A000:0000 is unavoidable for all graphics modes which need to address more than 256K of video RAM



However, a problem arises. Any mode that requires more than 256K per screen page cannot use standard methods for accessing video RAM. In the 16 color modes, the 64K segment starting at A000:0000 can be used to address only 256K of video RAM using the four bitplanes. This situation is even worse with the 256 color modes. These have only 64K available, although earlier in this chapter we showed a way to address the entire 256K (see "The VGA 256 Color Graphics Modes" information in the "EGA And VGA Cards" section).

Therefore, all Super VGA cards use a mechanism that allows the entire video RAM of the card to be addressed using the memory segment at A000H. Unfortunately, this access can only occur in 64K chunks for 256 color modes or in 256K chunks for 16 color modes. Various cards use different hardware registers to make a piece of the video RAM available at A000H. These registers aren't found on standard VGA cards. The same mechanism manages EMS memory (see Chapter 12 for more information).

Copy a section of video RAM to the 64K window at A000:0000 by dividing the offset address into page and offset components. This process doesn't simplify the routines to set or read a pixel, because you now must know which 64K memory window, as well as the offset in video RAM, to access the desired byte. You also must remember that your 64K window can be moved only in certain increments (granularities), such as 1K, 2K, 4K, 8K, etc. You can also think of the video RAM as being divided into "pages" under this scheme. The page size corresponds to the granularity of the 64K window. If you choose 1K, then the 64K window at A000H will have 64 pages. The number of the first page is set by a hardware register called the page select register.

The visible memory region at segment A000H cannot begin at just any offset address. It must be a multiple of the granularity. For example, suppose that you want to access the address 65539 ($64K + 3$) with a granularity of 1K. You would load 64 into the page select register. Then you could access the desired location at A000:0003. In this case, there are 63 other possible combinations for the value in the page select register and offset address for accessing the segment A000. Another example would be to load 63 in the page select register to access the byte at $1024 + 3$ relative to memory segment A000H.

Because of this scheme, graphics routines must convert offset addresses in video RAM to the combination of a page number plus the offset relative to memory segment A000H. This isn't a difficult task, once we consider the binary nature of these components.

The starting point is the offset for access to the video RAM, which must be 20 bits wide to address 1 megabyte of video RAM. If it's only 512K, then 19 bits will be sufficient. For our example, we'll assume 20 bits. If you're using a granularity of 1K, then this offset must always be divided by 1024 to obtain the value for the page select register. In binary terms, "divide" always means the dividend must be moved to the right the number of bits needed for binary representation of the divisor. Since 1K corresponds to a value of 210, the 20 bit offset address must be moved 10 bits to the right to obtain the desired number. The digits that are moved to the right of the "decimal" then become the offset in memory segment A000H.

The paging mechanism also has other tasks besides calculating page numbers and offsets. It becomes difficult when you must copy pieces of the video RAM that are more than 64K apart from one another. This makes it impossible to fit everything in the segment at A000 at the same time. So, the copy procedure must be divided into three time-consuming steps. First, the source is loaded into the segment at A000 so it can be copied to a buffer in main memory the program creates. Then the destination region must be loaded at A000 so the contents of the buffer can be loaded there.

This type of operation is fairly common, especially with animation applications. So, many Super VGA cards divide the memory segment at A000H into two 32K blocks, each of which has its own page select register. This allows the segment at A000H to handle two different regions of video RAM simultaneously. The regions can then be copied directly via the video RAM without having to use a buffer.

There are also no Super VGA standards for the process that controls page switching. Different cards use different registers and different granularities in segment A000H. Also, you cannot assume that all cards divide the A000H segment into two paging regions.

This causes a lot of problems for any programmer who wants to use a Super VGA mode that requires more than 256K. So it's obvious why the manufacturers of some programs, such as AutoCAD or Windows, let the card manufacturers develop software drivers for their products.

However, the VESA standard represents a solution to this problem. This standard represents a hardware independent interface for programming Super VGA modes. We'll discuss the VESA standard starting on the next page.

Doing it yourself

If, despite all the problems that may occur, you still want to write a program using a Super VGA graphics mode, you should use the 800x600 pixel mode with 16 colors for several reasons. This mode is available on most Super VGA card and it requires only 256K of video RAM. This eliminates having to program the page select register and will ensure that all Super VGA cards, regardless of how much memory they have, will be able to use your program. This mode will also allow you to use a lot of the routines that we've already seen work with the standard VGA 16 color modes.

Sample programs

We'll demonstrate how to do this in the following programs which you'll find on the companion CD-ROM. The Pascal (V8060P.PAS) and C (V8060C.C) versions are based on routines from "The 16 color EGA and VGA graphics modes" in the "EGA And VGA" section of this chapter. Initializing the 800x600 mode and setting or querying pixels are all accomplished with the various assembler routines found in the modules V8060CA.ASM and V8060PA.ASM.

The initialization routine INIT800600 performs the most difficult task. This routine must determine which code number the 800x600 pixel graphics mode uses on the installed video card. One way the routine can do this is by trying different code numbers until it finds one the BIOS accepts. There is a kind of array within the assembler program that stores six code numbers that are often used to identify the 800x600 pixel graphics mode. This array is called MODENO.

The most widely used code numbers are first put in the array. These code numbers are 6AH, 58H and 29H. The code numbers 54H, 16H and 79H are also in the array, but these are less common. We encountered these codes frequently while testing various Super VGA cards. They should work for most of the Super VGA cards on the market. If you find a particular card with a different code, you can easily add this code number to the MODENO array.

The INIT800600 routine recognizes the proper code number after it uses one of the codes to successfully initialize a graphics mode with function 00H of the video BIOS. Function 0FH of the video BIOS is used to return the code number of the currently active video mode. If this code number doesn't match the code number the program sent to initialize the mode, then the desired mode wasn't initialized and another code must be tried.

As long as the call to function 0FH doesn't return a match of the code number, the INIT800600 routine will continue to run through the array. This occurs until a code is accepted by the BIOS or until all codes in the array have been tried. In this case, the routine returns a value of 0 to its caller in the C version or a value of FALSE in the Pascal version. If the mode could be successfully initialized, however, the C version returns 1 and the Pascal version returns TRUE.

According to our tests, this method can be used to set the proper video mode with several Super VGA cards. However, this method isn't completely foolproof; the entire process can fail if the video card uses one of the codes in the MODENO array for a video mode other than the 16 color 800x600 pixel mode. In this case, the INIT800600 routine will return a value of 1 or TRUE, but the rest of the program won't work properly because a different video mode than expected will be functioning. However, this didn't occur in any of our tests, so this doesn't seem likely.

You'll find the following program(s) on the companion CD-ROM



V8060P.PAS (Pascal listing)
V8060PA.ASM (Assembler listing)
V8060C.C (C listing)
V8060CA.ASM (Assembler listing)

The VESA standard

The history of the PC has shown hardware manufacturers how the lack of standards can hurt business. Without standardization, manufacturers can develop features that give their products advantages over the competition. However, a video card that has great features but cannot work with any programs is useless. Software developers aren't going to risk developing products that will work only on one manufacturer's video card. This has been the case in the Super VGA market. Manufacturers are offering different cards with different features and no standard methods for addressing video RAM or

using the hardware registers. The result is that software developers have been very reluctant to produce programs that use the expanded Super VGA modes. The card manufacturers must therefore deliver software drivers for their cards so at least some programs can use the additional modes.

In an attempt to solve this problem, several VGA manufacturers established the VESA committee (Video Electronic Standards Association) in 1989. ATI, Chips & Technologies, Everex, Genoa, Intel, Phoenix Technologies, Orchid, Paradise, Video Seven and many other companies are part of this association.

This committee's goal was to develop a BIOS expansion that would give programmers hardware independent access to the Super VGA features found on many manufacturers' cards. This BIOS expansion, called the VESA standard, was released in 1990. The BIOS expansion described by this standard will be incorporated in the expanded VGA-BIOS found in the ROM chips on these manufacturers' cards. There will be software drivers for older cards. These drivers will be run as TSR programs to add the VESA functions to the existing BIOS. For the first time, this will give programmers the freedom to develop applications that work with many Super VGA cards, without having to consider the various hardware differences.

The VESA standard graphics modes

First, the VESA committee members had to decide which graphics modes to support. Obviously each manufacturer wanted their own modes in the standard. After some compromise, they came up with a list of nine modes, which are listed in the following table.

Code numbers were assigned to the graphics modes so the VESA BIOS could identify them the same way the normal VGA BIOS does. Codes over 100H (256) were selected, however, because the individual manufacturers had already used many of the codes under 100H for their own special modes. So, Super VGA modes are represented with 16 bit codes under the VESA standard. The one exception to this is the 16 color 800x600 pixel mode, which many manufacturers had already assigned the code 6AH. The VESA standard keeps this code for this mode and supports it as one of the expanded VESA modes.

The VESA BIOS Graphics Modes							
Code	Resolution	Colors	Memory*	Code	Resolution	Colors	Memory*
101H	640x 480	256	512K	105H	1024x 768	256	1 Meg
102H	800x 600	16	256K	106H	1280x1024	16	1 Meg
103H	800x 600	256	512K	107H	1280x1024	256	1.25 Meg
104H	1024x 768	16	512K	6AH	800x 600	16	256K
* This number is given in increments of 256, 512 or 1024K and does not represent the actual memory needed for a single screen page.							

Calling the VESA BIOS functions

The VESA BIOS functions can be accessed with the BIOS video interrupt 10H, like the normal functions of the VGA BIOS. There are six subfunctions under function 4FH. Before the function call, the function number 4FH must be placed in the AH register and the desired subfunction number (0 - 5) must be placed in the AL register.

The AH and AL registers are also used to read the results of the function after it has been called. If the VESA BIOS functions are supported by the installed video card, then the value 4FH will be returned to the AL register. The AH register receives a status code. A value of 0 indicates the function was successfully executed and a value of 1 indicates that something went wrong. All other return values (01H - 0FFH) are reserved, but should be handled as error messages if they are encountered. The contents of the registers other than AH and AL aren't changed by these function calls, unless they are used to return specific information.

The VESA BIOS functions	
No.	Description
00	Query capabilities of Super VGA card
01	Query identifying information for a certain mode
02	Set a VESA mode
03	Query current video mode
04	Save/restore status of the Super VGA card
05	Set/query access window to video RAM

Usually it isn't necessary to query the contents of the AH and AL registers after every VESA function call to check for errors. But you should always check the results returned to the AH and AL registers after your first VESA function call to ensure the VESA functions are supported by the installed video card.

Checking a Super VGA card's capabilities

Although the VESA standard provides an interface to the most important video modes of the new Super VGA cards, there is no guarantee that a particular card will support all the VESA modes. So, at run time (before working with any of the VESA functions), you must call subfunction 00H so the capabilities of the installed Super VGA card can be queried. For this function call, the function and subfunction numbers are expected in the AH and AL registers, respectively. A FAR pointer to a 256K buffer is also expected in the ES:DI register pair. Subfunction 00H stores information about the capabilities of the installed Super VGA card in this buffer.

Structure of the Subfunction 00H Buffer					
Ofs.	Contents	Type	Ofs.	Contents	Type
00H	VESA Signature ("VESA")	4 BYTE	06H	FAR pointer to an ASCII string containing the name of the card manufacturer	1 DWORD
04H	VESA Version, main version number	1 BYTE	0AH	Flag that indicates the capabilities of the card currently not used, therefore 0000h	1 DWORD
05H	VESA Version, secondary version number	1 BYTE	0EH	FAR pointer to the list of code numbers for the supported video modes	1 DWORD

The most important information for the programmer is obtained via the last pointer in this table. This points to a list of the code numbers, for the video modes, supported by the installed card. This list may be either in the ROM-BIOS of the card or in main memory. It consists of several words. Each word indicates the number of a supported mode. Both the standard VESA modes and the manufacturer's custom modes are listed. The manufacturer's own modes can be easily distinguished from the VESA modes because their code numbers are less than 100H and therefore contain a value of 00H in the high byte of the corresponding entry in this list.

This list can be a different length for each card, depending on the modes it supports and how much RAM it has. The end of the list is indicated by a word containing the value FFFFH, which doesn't represent any video mode.

Reading a specific video mode

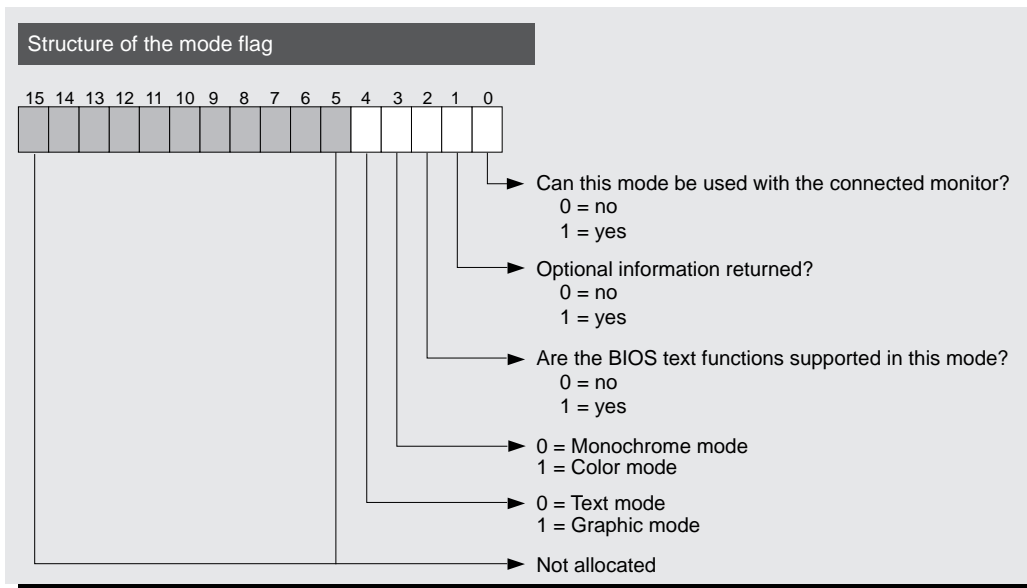
Just knowing that a certain mode exists doesn't mean that you can use this mode. Subfunction 01H is used to query all the information needed to program a given video mode. The function and subfunction numbers are expected in registers AH and AL, respectively. Also, the number of the appropriate mode must be passed in the CX register. This number must correspond to one of the entries in the list that is queried with subfunction 00H. The ES:DI register pair expects a pointer to a buffer that will store this information. This buffer must be able to store 29 bytes.

The subfunction 00H mode list includes both the VESA modes and the manufacturer's own modes. This allows you to use subfunction 01H to retrieve information about the expanded Super VGA text modes as well as the standard VESA modes.

The following table shows the structure of the information as it's entered in the buffer by subfunction 01H.

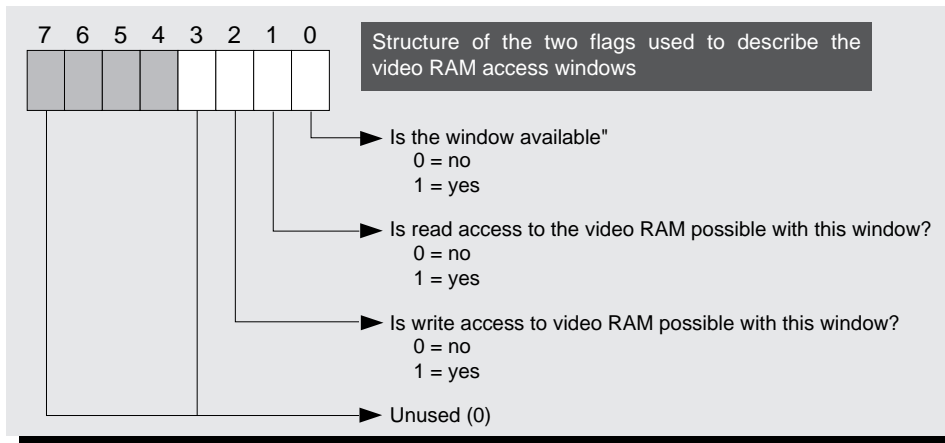
Structure of the subfunction 01H buffer		
Ofs.	Contents	Type
00H	Mode flag, see below	1 word
02H	Flags for the first access window, see below	1 byte
03H	Flags for the second access window, see below	1 byte
04H	Granularity of the two access windows, in K,	1 word
06H	Size of the two access windows, in K	1 word
08H	Segment address of the first access window	1 word
0AH	Segment address of the second access window	1 word
0CH	FAR pointer to routine for setting the visible region in the two access windows	1 dword
10H	Number of bytes required by each pixel line in video RAM word	1 byte
Optional information, see mode flag		
12H	X resolution in pixels/character	1 word
14H	Y resolution in pixels/character	1 word
16H	Width of character matrix in pixels	1 byte
17H	Height of character matrix in pixels	1 byte
18H	Number of bitplanes	1 byte
19H	Number of bits per screen pixel	1 byte
1AH	Number of memory blocks	1 byte
1BH	Memory model	1 byte
1CH	Size of memory blocks in K	1 byte

The mode flag (the first bit field in the block) provides information on the queried mode. As shown in the following figure, the mode flag indicates if the mode is a text or graphics mode, if color is supported, if text can be output with the BIOS functions and, most importantly, if this mode can even be used with the available monitor and memory.



It's also important to determine whether the information in the optional fields of the data block are filled. These fields contain important information for working with a given video mode. This information is usually available.

The two flags that describe the access windows to the video RAM are also represented by bit fields. Since not all Super VGA cards have two access windows, these flags must be used to determine whether a second window is available. Also, these flags will indicate which window is for reading and which is for writing.



In addition to the flags that describe the video RAM access windows, some other important information about using the access windows is also given in the subfunction 01H data block. This includes the segment addresses of the two access windows. Obviously the entry for the second window is only valid if the second window actually exists. The size and the granularity of the two windows are also given. Remember the granularity is the interval with which the window can be "moved" through the video RAM.

The VESA-BIOS has a routine for moving the window through video RAM. This routine protects programmers from the incompatibilities between different Super VGA cards. You'll find more information about this routine when we discuss subfunction 05H at the end of this section.

The optional information in the block begins with the X and Y resolution of the video mode. If the mode is a text mode, this information will pertain to text lines and columns; otherwise, the resolution is given in pixels. The next two fields, which give the size of the character matrix in pixels, are used only for text modes.

The number of bitplanes used by the mode and the number of bits required to code a single pixel are listed after these fields. These two fields are only required for graphics modes. The next field, which gives the number of memory blocks, is only required for use with the graphics modes of CGA and Hercules graphics cards. This is because these cards store graphics lines

in video RAM using memory blocks of different sizes. The last field of the data structure gives the size of the memory block used. Before this field, however, the memory used for direct access to video RAM is given. The memory models are coded as listed in the table on the left.

Valid codes for describing memory models	
Number	Description
00H	Text mode
01H	CGA format, 2 or 4 memory blocks
02H	Hercules format with 4 memory blocks
03H	Normal EGA/VGA format for 16 color graphics modes
04H	Compact format - 2 pixels (4 bits each) per byte
05H	Normal EGA/VGA format for 256 color graphic modes
06H-0FH	Reserved
10H-FFH	Manufacturer-specific codes, currently unused

Setting a mode

Once you've used subfunction 00H to activate the list of supported video modes and subfunction 01H to read the

requirements of the various modes, you can select the mode that meets your program's needs. Then you can use subfunction 01H to set this mode.

Before the function call, the function number and subfunction number must be supplied as usual. Also, the code number of the desired mode must be placed in the BX register. If you want to keep the currently existing contents of the video RAM when the new mode is set, bit 15 of the BX register must be set to 1. After the function call, you should always check to ensure the correct mode was set. To do this, read the contents of the AH and AL registers. If you find values of 00H in AH and 4FH, then the mode was successfully set.

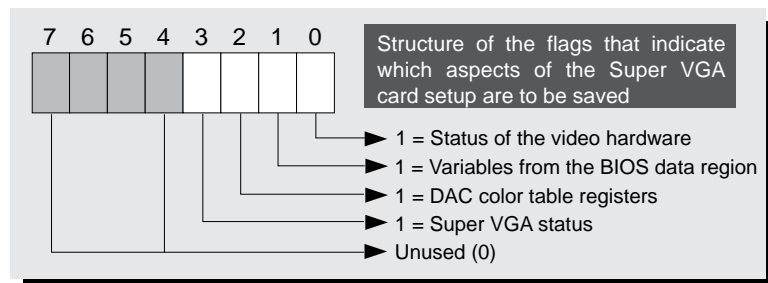
Subfunction 03H can be used to query the current video mode. This function call doesn't require any arguments besides the function and subfunction numbers. After the function has run, the number of the current video mode can be read from the BX register. As usual, values greater than 100H are for the standard VESA modes and values less than 100H are for standard VGA or manufacturer-specific modes.

Saving and restoring a setup

When TSR programs such as SideKick are activated, they must save the current setup of the video card before they can switch to the mode they will use. Also, these programs must be able to restore the contents of the screen to its original state. Some of the video RAM must be saved to accomplish this. This is very difficult to do with Super VGA cards. So, the VESA standard has three sub-subfunctions under subfunction 04H that help a program manage these tasks.

Subfunction 00H must be called before the current video card setup is saved because this function indicates how much memory is needed to save this information. In addition to the function and subfunction numbers, this function also requires the value 00H (sub-subfunction number) in the DL register. In the CX register, a bit field, which describes the components of the video card setup that must be saved, is needed.

The structure of this bit field is shown in the illustration directly above.



The entire contents of the video RAM doesn't have to be saved. Only the portion overwritten by the TSR program must be saved. For example, a TSR program that switches from a graphics mode to a text mode must save only the first 4K of video RAM, which will be overwritten for displaying text. It's not necessary to save the entire graphics screen, which could take up to 1 megabyte of memory.

As a result, sub-subfunction 00H returns the number of 64 byte blocks, which are needed to save the indicated status information, to the BX register. The calling routine must prepare a buffer to store this information. The size of the buffer to create is $64 * BX$ bytes.

Once a program has created this buffer, the setup can be saved with sub-subfunction 01H. This function call requires three parameters in the processor registers in addition to the function and subfunction numbers. The DL register expects the sub-subfunction number 01H. The CX register must contain a bit field that describes which components of the setup should be saved, as we saw with sub-subfunction 00H. It's important that this bit field doesn't contain more components than that of the 00H sub-subfunction call; otherwise the buffer may not be large enough.

The address of the buffer itself must be passed to sub-subfunction 01H as a FAR pointer in the ES:BX register pair.

The same registers are used to restore the saved setup. This is done with a call to sub-subfunction 02H. For this function call, the sub-subfunction number is placed in the DL register. The CX register again must contain a bit field with the components of the setup to be restored and the ES:BX register pair must contain a FAR pointer to the buffer with the stored information.

Moving the access window

Since the entire video RAM can be accessed through the 64K segment at A000:0000, you can easily work with Super VGA modes that use less than 256K of memory per screen page. This process becomes more complicated when more than 256K must be addressed, which applies to most of the modes we're discussing. In these instances, you must use access windows. These windows allow you to use the 64K segment at A000:0000 (together with the four bitplanes) to access a total of 256K of video RAM simultaneously. Earlier we saw that setting the access window involves programming registers that aren't standardized in Super VGA cards.

The VESA-BIOS solves this problem by providing subfunction 05H. This function is used to set up the access window independent of the hardware on your Super VGA card. In addition to the function and subfunction numbers, this function call requires the value 00H in the BH register and the number of the access window (either 0 or 1) in the BL register. Remember the second access window can be moved only if the subfunction 01H call has indicated the installed video card actually has a second access window.

The last parameter is placed in the DX register. This is related to the granularity of the access window and indicates the start of the addressable region in video RAM. For example, if the granularity is 1K, then a value of 256 would make the second 256K of the video RAM on a Super VGA card available using the 64K segment at A000:0000.

This function call is often needed in the high resolution Super VGA modes. Since calling it as an interrupt function would take too much time, the VESA-BIOS calls this function directly with a FAR call. Subfunction 01H returns the address of this routine as a FAR pointer in its data block. Remember, support of the FAR call method isn't necessary for every VESA BIOS, in which case the address of the routine would be returned as 0000:0000.

But if subfunction 05H is available as a FAR call routine, then you should definitely call it if you're concerned with your program's execution speed. One difference between the FAR call and the interrupt call is the FAR call doesn't return a status code to the AX register. This method still reserves the ability to change the contents of the AX and DX registers during a function call.

Subfunction 05H can also determine the position of the access window in video RAM. To call the function in this way, you must start with the function number and subfunction number as usual. The value 1 must also be entered in the BH register. As a result, the DX register will contain the location of the window, in relation to its granularity.

5

The Interrupt Controller

Like other components of the computer system, the CPU is scarce resource. If it is busy performing one task, then it can't handle another. To relieve the CPU of some of the frequent and routine tasks associated with input and output devices, the original PC design includes an interrupt controller.

When first used by Intel in 1985, the 8259A Programmable Interrupt Controller, more commonly known as the PIC, is found between the CPU and the peripheral devices. The controller has since been superseded by "chip sets" which also handle additional chores which were previously performed by other discrete components.

Nonetheless, the function of the 8259A and its successor remains the same. The PIC is the computer's connection to the outside world. Interestingly enough, the 8259A has several capabilities which are not completely used by the PC - namely the "rotation of priorities", which we'll talk about later. The new replacement chip sets do not have this capability.

The inner life of the interrupt controller

The interrupt controller acknowledges an interrupt from a peripheral device and passes it on to the CPU. The CPU can then activate the corresponding program or routine to handle this particular interrupt. A computer system has many peripherals, all of which require handling by the CPU. The keyboard, hard drive, printer and timer all require attention of the CPU. But at any given moment, the CPU can attend to only one at a time. The interrupt handler's job is to present the various interrupts to the CPU in sequence. Some peripheral devices must be serviced quickly and others can tolerate a slight delay.

The PIC can handle eight interrupts. These are assigned a number ranging from 0 to 7. These numbers also represent the interrupt priority. IRQ0 has the highest priority while IRQ7 has the lowest priority. There are no standards which dictate which interrupt sources are connected to a given PIC interrupt line. These are determined by the board designer.

The table on the right lists the eight interrupt priorities of the original IBM-PC/XT. Note that the timer has a substantially higher priority than the printer and the keyboard is subordinate to the timer.

One of the special features of the PIC is its ability to "cascade" interrupts. If more than eight different interrupt sources need to be served, you can cascade or combine multiple PICs. Since the early days of the PC/AT, this method has been used so 16 different hardware interrupt devices can be handled.

Interrupt	Device
IRQ0	Timer
IRQ1	Keyboard
IRQ2	Free
IRQ3	Second serial interface
IRQ4	First serial interface
IRQ5	Hard drive
IRQ6	Disk drive
IRQ7	Printer

AT Hardware Interrupts			
Priority	Interrupt master	Slave	Device
(highest) 0	IRQ0		Keyboard
1	IRQ1		Keyboard
2		IRQ0	Realtime clock
3		IRQ1	Unused
4		IRQ2	Unused
5		IRQ3	Unused (i.e., sound card or CD-ROM drive)
6		IRQ4	Unused
7		IRQ5	Math coprocessor
8		IRQ6	Hard drive
9		IRQ7	Unused
10	IRQ2		Realtime clock
11	IRQ3		Second serial interface (COM2:)
12	IRQ4		First serial interface (COM1:)
13	IRQ5		Second parallel interface (LPT2:)
14	IRQ6		Hard drive
(lowest) 15	IRQ7		First parallel interface (LPT1:)

If several PICs are operating together, one always acts as the master and the others are slaves. Only the master is connected directly to the CPU. The slaves pass their interrupt demands to the master. We'll see how this works shortly.

The PIC connections

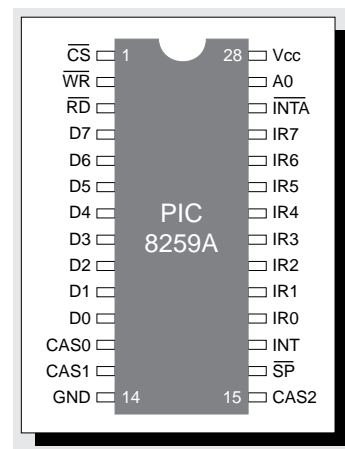
To understand how the PIC works and how it interacts with the CPU, let take a look at the connections. The original PIC, the Intel 8259A is connected to other components through its 28 pins (see illustration to the right). An interrupt source is connected to one of the eight pins identified as IR0 through IR7. To trigger an interrupt, the source sets the pin to high.

Based on its priority and whether another interrupt is already in progress, the PIC determines if it can transfer the interrupt to the CPU. If so, it sets INT to high. This line is in turn connected to the processor's INTR pin and signals that a hardware interrupt has occurred. If the CPU is ready to accept the interrupt, it responds to the PIC on pin INTA. The PIC in turn responds by sending the interrupt number over pins D0 through D7. These lines serve as the data bus for transmitting the individual bytes of information between the PIC and the CPU.

Following this dialog, the CPU is now ready to call the appropriate interrupt handler. The PIC interrupt number is an index into the table of interrupt vectors. The corresponding interrupt handler is selected by simply extracting its address from the table.

On the hardware side, the communication between the CPU and the PIC is finished for the time being. When the interrupt handler has completed its work and before it returns control to the CPU, the interrupt handler contacts the PIC. In doing so,

INTEL 8259A connection usage



the PIC can then allow other interrupts access to the CPU. The exact procedure is discussed in a later section on PIC programming. The following table describes the line on the PIC.

Wire	Pin	Name	Task
-CS	1	Chip-Select	Used in connection with -RD and -WR signals for communication between the PIC and CPU over the data bus.
-WR	2	Write	If this signal is low, the CPU can write data over the data bus to the internal PIC register.
-RD	3	Read	If this signal is low, the CPU can read data from the internal PIC registers over the data bus.
D0-D7	4-11	Data bus	Data is transferred between the PIC and the CPU over these pins.
CAS0-CAS2	12,13,15	Cascade	A master PIC communicates with its slave over these pins.
GND	14	Ground	Ground
-SP	16	Slave Program	Indicates if the PIC is a master or slave.
INT	17	Interrupt	This pin is connected to the CPU INTR pin and signals a hardware interrupt to the CPU.
IR0-IR7	18-25	Interrupt request	Each line is connected to an interrupt source, which signals an interrupt. IR0 indicates the interrupt IRQ0 and is of the highest priority, IRQ7 represents the lowest.
-INTA	26	Interrupt Acknowledge	Indicates that that the CPU is ready to accept the interrupt number from the PIC.
A0	27	Address	Register number to be used during read and write.
Vcc	28		Voltage source (depending on the system +3.3 or +5 Volt)

To keep track of which interrupt is being serviced and which is waiting to be serviced, the PIC has the following three internal registers:

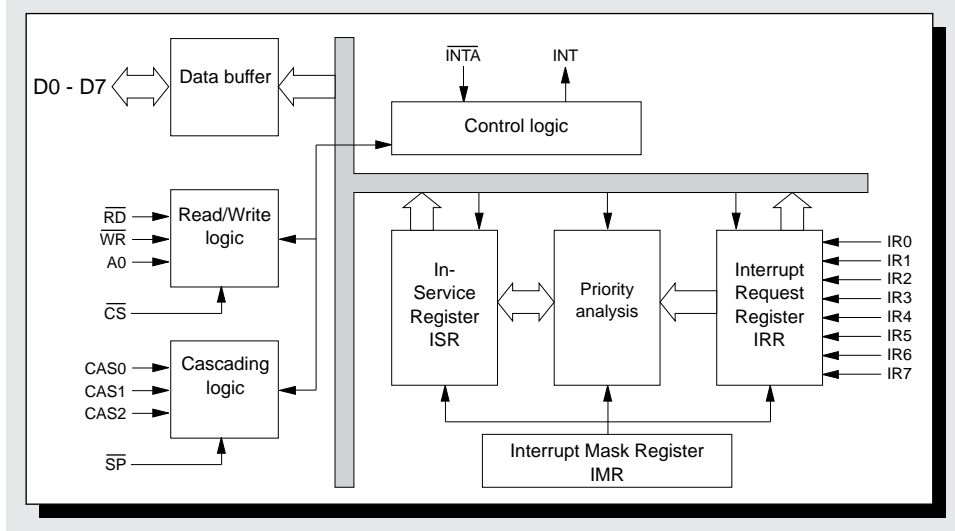
- Interrupt Request Register (IRR)
- In Service Register (ISR)
- Interrupt Mask Register (IMR)

These three registers are all eight bits wide. Each bit corresponds to one of the eight hardware interrupts. The Interrupt-Request-Register is connected directly to the eight interrupt lines - IR0 through IR7. To request an interrupt, the interrupt source sets its line to high. The corresponding bit in the IRR is simultaneously set. In this way, the PIC knows which source requested the interrupt. The Interrupt Mask Register can be used to mask or ignore one or more interrupt sources. If a bit in the IMR is set, then an interrupt from the corresponding interrupt source is ignored.

For non-masked interrupts, the PIC has to determine if it should present that interrupt to the CPU. Since additional interrupts may be awaiting service, the PIC selects the bit from the IRR having the lowest value and therefore the highest priority. This is compared to the contents of the ISR which indicates the source of the most recent interrupt. If the IRR has a higher priority interrupt awaiting service, then the PIC passes this onto the CPU - a process which effectively preempts the lower priority interrupt. The PIC continues this watchdog operation until all interrupts having a higher priority have been serviced.

When the PIC passes an interrupt onto the CPU, it sets the corresponding bit in the ISR register. At the same time the corresponding bit is cleared from the IRR register. This way, it keeps track of which interrupt is being handled and won't try to service this interrupt again.

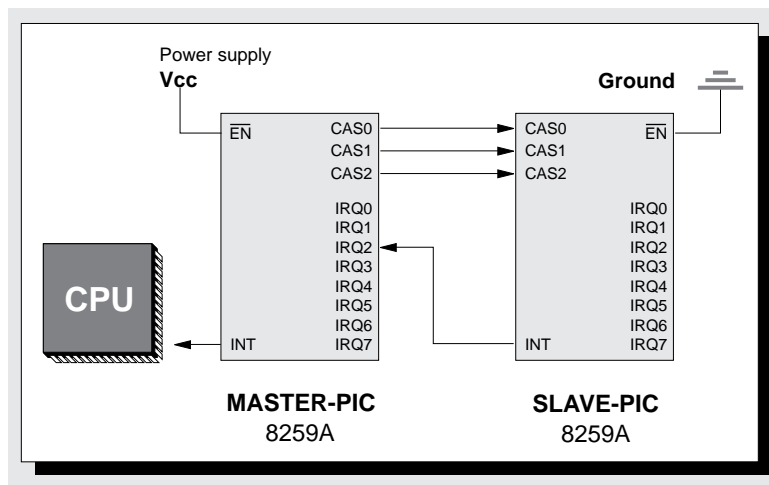
Block diagram of the PICs with the internal registers IRR, ISR and IMR



Cascading

Recall that a single PIC is limited to handling only eight interrupt sources. To overcome this limitation, "cascading" is used. Simply stated, cascading is a way of linking together several PICs. Line EN determines if the PIC operates as a master or a slave. When voltage is applied to line EN, the PIC acts as a master. Otherwise, it functions as a slave. Three of the slave's lines (CAS0, CAS1 and CAS2) are connected to the corresponding lines on the master and the INT line on the slave is connected to the IRQ2 line of the master.

Connecting (switching) two PICs as master and slave



Interrupts on the master that do not involve cascading are handled the same as described above. Let's see how they're handled on a slave. We'll talk about the similarities first. Devices are connected to one of the slave's eight IRQx lines. Based on the interrupt's priority and whether another interrupt is in progress, the slave determines whether it can pass this interrupt onto the CPU. If so, the slave sets INT to high.

Now we must talk about the differences. By setting INT to high, the slave is actually triggering IRQ2 on the master. The master recognizes that a slave is attached and treats the interrupt the same as if it originated directly from the hardware device.

IRR register bit 2 is set in the master. If the IMR and ISR registers allow it, the interrupt is passed onto the CPU. Concurrently, it sets the number of the slave that sent the interrupt request on its lines CAS0, CAS1 and CAS2. These lines are connected to the lines of the same name on the slave.

Since CPU line INTA is connected to both the master and the slave, the CPU lets the slave know that it's ready to accept an interrupt by setting line INTA. The slave passes the interrupt number onto the CPU, which then initiates the call of the appropriate interrupt handler. Both master and slave then clear the respective bit in their internal IRR registers and set the corresponding bit to one in the ISR registers to keep track of interrupts in progress. When the PC is first booted, the BIOS initializes the registers of both PICs so they work correctly.

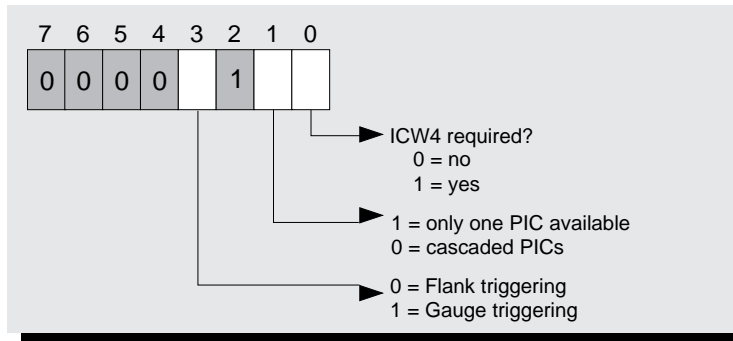
Initializing a PIC

As its name suggests, the PIC is programmable and responds to various commands. Basically there are two categories of PIC commands: initialization commands and operational commands. In fact, these categories have their own acronyms: ICW for Initialization Command Words and OCW for Operational Command Words. ICWs are sent to the PIC in a very precise order since they're interdependent. OCWs can be sent to the PIC in any order.

Two port addresses are used to talk to the PICs. The port for the master is located at 20h and 21h; the port for the slave is at A0h and A1h. Initialization always begins by writing the first initialization command word, ICW1 to port 20h or A0h, depending on whether the PIC is to be initialized as master or slave.

ICW1 is composed of eight bits. Only certain bits are significant to this discussion: ICW1 bit 0 specifies whether three additional initialization command words are to be sent to the PIC. These are ICW2, ICW3 and ICW4. If bit 0 is set, then the PIC can expect these three additional ICWs.

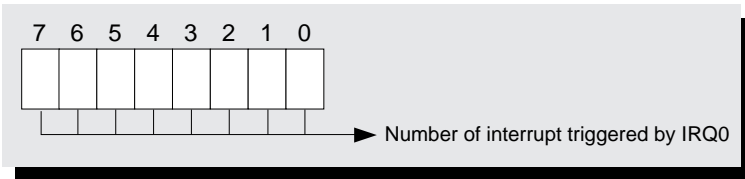
ICW1 bit 1 specifies whether the PIC is to be used alone or if several PICs are to be cascaded. If bit 1 is clear, then several PICs are to be cascaded. ICW1 bit 3 specifies how PIC lines IRQ0 to IRQ7 respond to the source devices. If bit 3 is clear, the PIC responds as an pulse-triggered device in which the interrupt is triggered by briefly setting IRQx to high and then letting it return to low. If bit 3 is set, the PIC responds as a edge-triggered device in which the interrupt is triggered by setting IRQx to high and leaving it there until the interrupt source drops it to low.



The BIOS sets ICW1 to the value 00010001b. This means that three additional initialization control words are to be sent, that cascading PICs are to be used and that flank triggering is to be used:

```
mov    al,00010001b
out    20h,al;ICW1 sent to master
out    0A0h,al;ICW1 sent to slave
```

ICW2 always follows ICW1. ICW2 defines the base address for the first interrupt (IRQ0). The second interrupt (IRQ1) then triggers the interrupt at the base address +1; the third interrupt (IRQ2) triggers the interrupt at the base address + 2, etc.



The BIOS sets the base address in ICW2 to 08h for the master and to 70h for the slave.

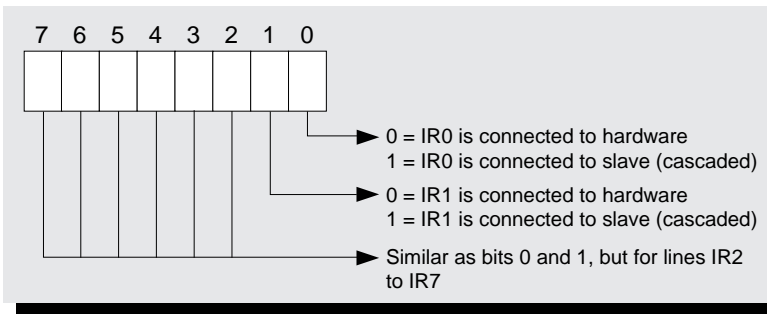
This means that hardware interrupts handled by the master activate the interrupt handler for the interrupts 08h to 0fh. Hardware interrupts handled by the slave activate the interrupt handler for interrupts 70h to 77h. It's important that ICW2, ICW3 and ICW4 be sent over the second PIC port.

```

mov  al,08h      ;08h is base interrupt for the master
out  21h,al      ;send ICW2 to the master
mov  al,70h      ;70h is the base interrupt for the slave
out  0a1h,al     ;send ICW2 to the slave

```

ICW3 follows ICW2 and must be sent to the PIC if the PICs are to be cascaded. If ICW1 bit 1 is clear, then the PIC expects ICW3. ICW3 has the task of informing master and slave through the interrupt control, to which they are connected. The ICW3's task depends on whether it is directed to the master or to the slave. The following appears with the master appears:



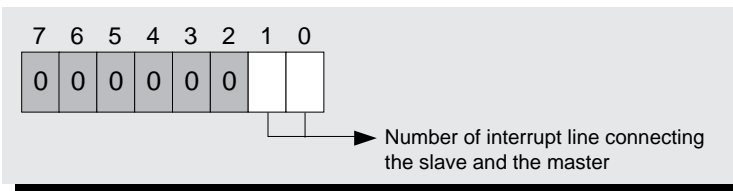
For the master, each bit in this register corresponds to one of the interrupt lines from IRQ0 to IRQ7, regardless of whether the line is connected directly to the hardware or to a slave. The BIOS sets only bit 2 to indicate that a slave PIC is cascaded over line IRQ2.

```

move  al,00000100b      ;cascading over IR2
out   21h,al            ;ICW3 send to the master

```

For the slave, ICW3 contains the interrupt control number.



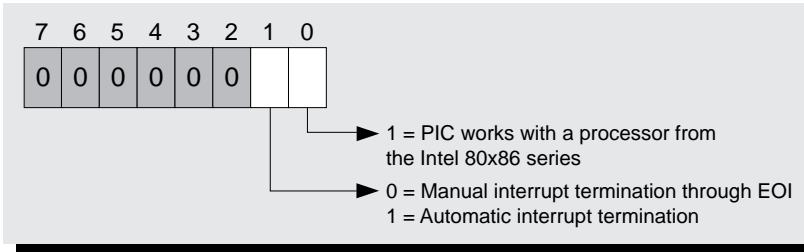
The BIOS sets ICW3 for the slave to the value 2.

```

mov  al,2          ;cascading over IR2
out  0A1h,al       ;ICW3 send to the slave

```

ICW4 indicates how the end of an interrupt is to be handled. A PIC can be programmed to automatically signal an end of an interrupt or to signal it by software. In a PC, the PIC requires software assistance, so bit 1 is set to 0. This has certain implications for the interrupt handler, which we'll talk about soon.



```

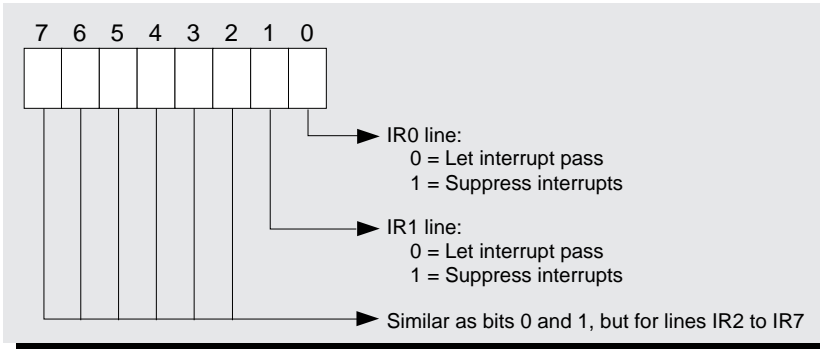
mov    al,00000001b    ;Intel environment, manual EOI
out    21h,al          ;send to the master
  
```

Control and inquiry of PICs - OCWs

As a rule, ICWs are sent only to the PIC once - from the BIOS. However user programs can obtain status information from or make changes to a PIC. To do this, the program sends OCWs to the PIC.

A PIC distinguishes between the various OCWs by their layout and the port over which they are sent.

OCW1 lets you change the PIC's Interrupt-Mask-Register. Changing the contents of the IMR lets you suppress individual hardware interrupts or reactivate them at a later time. If a bit is set to 1, the corresponding IRQx line is masked. When an interrupt line is masked, the device attached to that line cannot trigger an interrupt until it is cleared.

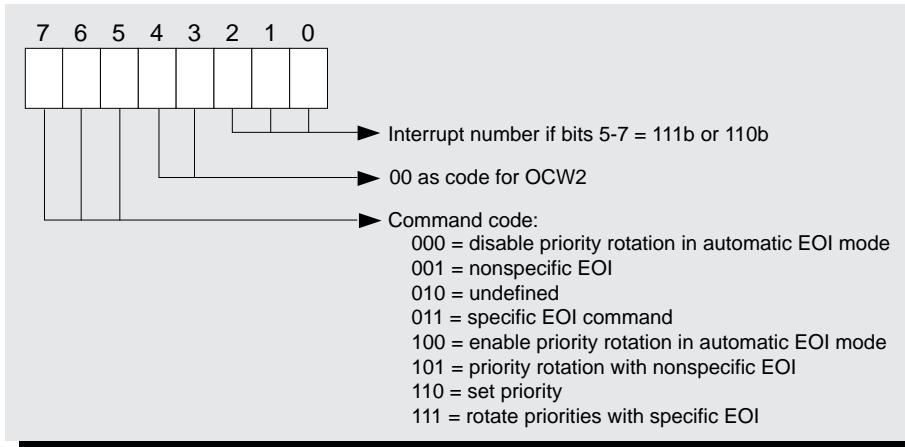


OCW1 is sent to the second PIC port - port 21h for the master and port A1h for the slave. Normally, all interrupts are enabled and therefore none of the interrupts are masked, so the value 0 is used here.

In the PIC master/slave combination, all slave interrupts may be suppressed if the bit 2 of the master's IMR is set to 1. The actual contents of this register can be obtained at any time by reading port 21h for the master or port A1h for the slave.

OCW2 is primarily used to signal the end of an interrupt handler. It also has other features such as priority rotation and automatic EOI handling. For PC's, neither of these features are enabled, so we'll talk about them only briefly in the following section.

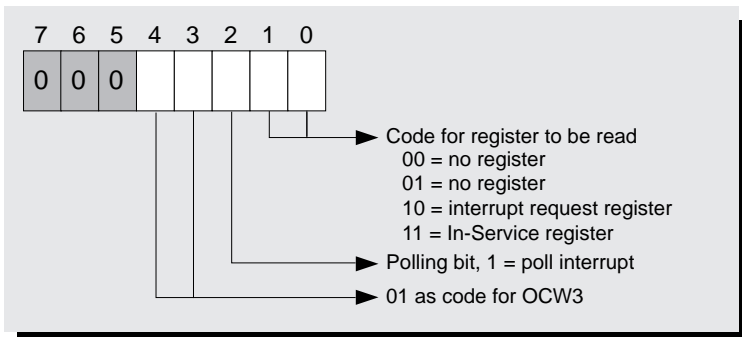
The PIC recognizes OCW2 only if it is sent to the first PIC port, either 20h for the master or A0h for the slave.



Of the various commands that can be sent using OCW2, only one is relevant for PC programming - code 001b. This is used to indicate the end of an interrupt. We'll talk more about this in the following section.

OCW3 lets you select the internal IRR and ISR PIC registers. Like OCW2, this command is sent to the first port. The PIC distinguishes between OCW2 and OCW3 by the setting of bits 3 and 4. For OCW3, these bits are set to 01b; for OCW2 they are 00b.

By writing the register number to the first port, the status of the desired register can be queried in the next step by reading the port. Therefore, a program can determine which interrupt handler is being serviced at that time and which are still awaiting service.



The polling bit has a special attribute if the PIC is operated in polling mode rather than interrupt mode. In polling mode, the INT PIC control is not connected to the INTR processor control. Therefore the PIC is not automatically notified of interrupts. Instead, the operating system ensure that the CPU checks the PIC status on a regular basis so as to recognize pending interrupt requests. With PC's, polling mode is not used.

Communication between the PIC and an interrupt handler

When the CPU sets line INTA, this tells the PIC that the interrupt has been accepted for processing by the CPU. But when the processing has been completed by the interrupt handler, the PIC has to be notified before it executes the IRET instruction.

The interrupt handler notifies the PIC by writing an OCW2 command to the first PIC port. The command code, 0001b means "unspecific EOI", where EOI stands for "End-Of-Interrupt". When combined with the other bits, the resulting command code is 20h.

For interrupts that are handled directly by the master, the corresponding machine language routine reads as follows:


```
mov    al,20h;EOI  command
out    20h,al;send to master
```

For a slave, the interrupt handler must also notify the slave PIC that "its" interrupt has also ended. The corresponding machine language routine for this reads as follows:

```
mov    al,20h;EOI  command
out    0A0h,al      ;send to slave, and also
out    20h,al;to master
```

Unused PIC capabilities

We mentioned a PIC feature called *priority rotation*. The purpose of priority rotation is to give the each line on the PIC an equal chance of service over an extended period of time. After a line is serviced, its priority is automatically dropped to the lowest level, while the ones at the lowest level are move up higher.

If the same interrupt happens again, it is serviced unless another interrupt with a higher priority isn't waiting for service. Although this method works well, it has an inherent problem when terminating the interrupt handler.

If an EOI command is passed to the PIC, normally it knows from which interrupt it originated, namely from the interrupt handler with the highest priority. But if the priorities can be changed, this simple arrangement no longer works. Besides the unspecified EOI, you can send a specific EOI to the PIC using OCW2, which contains the number of the respective interrupt. In this way, the PIC recognizes precisely which interrupt handler was terminated and can clear the corresponding bit in its ISR register. With PC programming, priority rotation is not used.

Sample program

The IRQSTAT program is an example of interrupt controller programming. This program, which you'll find on the companion CD-ROM, is designed to be a resident program that wedges itself into all hardware interrupts. You can execute many things from the DOS level, release the hardware interrupts, for example press the keyboard or access the floppy disk or hard drive. IRQSTAT notifies you of the respective hardware interrupts by displaying the status of the PIC's ISR and IRR registers in the upper right corner of the screen.

In this way, you can recognize which interrupts have just been executed and which are waiting to be executed. Because the interrupt is handled so quickly, the screen displays the information for only a short time. IRQSTAT displays a counter which is incremented for each hardware interrupt. In this way, you get a sense of when and how often a hardware interrupt is triggered.

IRQSTAT is based on the module IRQSTAT.C, which works with two additional modules: IRQUTIL.C and WIN.C. WIN.C is a collection of routines for the management of text windows and the display of text. IRQUTIL.C., on the other hand, is different. It contains routines for many important tasks, which fall into the same category as PIC management and query. Therefore, you'll be able to adapt the code from this module into your own programs. The table on the right lists the functions from the IRQUTIL.C module.

Function	Task
irq_Enable	Hardware-Interrupt admit
irq_Disable	Hardware-Interrupt conceal
irq_SendEOI	"End Of Interrupt" signal
irq_SetHandler	install new Interrupt-Handler
irq_ReadMask	read Interrupt-Mask-Register
irq_ReadISR	read In-Service-Register
irq_ReadIRR	read Interrupt-Request-Register

IRQSTAT works quite simply. The main program calls function InstallIRQ which redirects each of the 16 interrupts to the C-function IRQ by using function irq_SetHandler function. In other words, it creates a new interrupt-handler for all of the hardware-interrupts.

If InstallIRQ is called as the interrupt handler for the keyboard, timer or for one of the other devices, it then uses the PrintIRQ function. PrintIRQ displays the status of the PIC registers on the screen and the currently serviced interrupt number.

Because IRQ is called as the interrupt handler for all hardware interrupts, it must determine the interrupt for which it's currently performing its services. Otherwise, it would be impossible for it to call the original handler, which would quickly result in a system crash. PrintIRQ returns the number of the interrupt since it has to fetch the contents of the ISR register from which the current interrupt results.

Using this information, IRQ can determine the address of the old interrupt handler from a previously loaded array and then call it. Since the old handler is being accessed, IRQ doesn't need to transmit the EOI command to the PIC because that already occurs in the original handler.

To keep IRQSTAT as simple as possible we didn't build a deactivate feature. To deactivate IRQSTAT, you can simply reboot the computer. However, it's easier if you just start IRQSTAT in a Window's DOS box. This lets you deactivate IRQSTAT simply by closing the box. In any case, you'll be able to see that not all interrupts can slip through from Windows to a DOS box. For example, the timer, keyboard and disk drive can slip through from Windows to a DOS box but the hard drive interrupt cannot.

To install IRQUTIL.H in your own programs, you'll need files IRQUTIL.H and IRQUTIL.C.

You'll find the following program(s) on the companion CD-ROM



IRQUTIL.H (C listing)
IRQUTIL.C (C listing)
IRQSTAT.C (C listing)
WIN.H (C listing)
WIN.C (C listing)
WIN.PAS (Pascal listing)



The DMA Controller

The DMA controller has been a standard component in the PC from day one. DMA is an acronym for *Direct Memory Access*. DMA is either specialized circuitry or a dedicated microprocessor which transfers data between from memory and a peripheral device without using the CPU. Although DMA may periodically steal cycles from the CPU, data is transferred much faster than if the CPU has to transfer every byte.

The transfer is usually directed by a program which continuously reads a byte using the I/O port of the respective peripheral device into a CPU-register and writes it to the memory from there. Since the CPU is not involved in a DMA transfer, the speed of the transfer is increased. In fact, the DMA controller was a great performance feature in the early days of the PC because a disk drive, for example, delivered data faster than could be read by software in the 8088 or 8086 CPU.

Although the processing speed of the CPU is now 100 times faster than the original 8088, the DMA controller's performance ability has not kept pace. Like the 8088-CPU, it was originally made to operate at a clock speed of 4.77 MHz. Today's CPUs have achieved clock speeds of 100 MHz but the DMA controller has never surpassed 6 MHz. The original AT design use a clock speed of 3 MHz, which was slower than the one in the PC/XT. The DMA controller's decline began with the AT and it became less important.

Although the functional capabilities of the original DMA-controller 8237A still remain today, it no longer plays a major role in system design. Several PC add-ons, such as sound cards, for example, still support the DMA controller but can also read the data from the CPU through an I/O port. This method is considerably faster than using the DMA and which is why the DMA controller is now considered a "non-factor" in PC design.

Other DMA controller design weaknesses include the following:

- It limits the transfer to memory areas within the 64K size pages.
- It supports the 16-bit transfer only with a great deal of effort.
- It's never attained the level of performance for today's CPUs.

Therefore, the DMA controller is only used in system programming when communicating with hardware extensions, which accept or return data exclusively through DMA. These are becoming increasingly rare.

Hardware and software interaction during DMA transfers

One of the few peripheral devices that still uses the DMA controller is the disk drive. This is one of the best examples of hardware and software interaction during a DMA-transfer.

The starting point is using one of the operating system's BIOS functions for Interrupt 13h from a user program. These BIOS functions let you read or write one or more sectors from/to a floppy disk.

The BIOS function works in two phases: First, it commands the floppy disk controller to seek to the desired sector and read the contents of that sector into a buffer. Concurrently, the BIOS "programs" the DMA controller and the channel connecting the hardware to the disk drive. This program instructs the DMA controller where in main memory the data is to be transferred to and how many bytes are to be transferred.

When the floppy disk controller finds the sector, it activates the line to request that the DMA controller start the transfer. The DMA controller accepts the specified number of bytes from the floppy disk controller and transfers them sequentially into the buffer beginning at the main memory address programmed through the BIOS function.

During this process the DMA controller assumes control of the PC bus. The CPU remains uninvolved. The DMA controller directs the bus controls so the individual bytes are transferred directly from the disk drive to main memory without having to temporarily store them in a DMA controller internal register. Once all data has been transferred, the DMA controller again releases the bus, while the disk controller simultaneously executes a hardware interrupt.

Through this hardware interrupt, the BIOS is notified that the data is now available in the designated buffer and returns to the BIOS function.

The DMA controller, however, is not only able to transfer data from a peripheral device into memory (and in the opposite direction), it's also able to copy memory areas within the main memory. It's unfortunate the original design of the DMA limits the transfer to a 64K block. For that reason, many modern PC's no longer support memory-to-memory transfers using DMA.

To eliminate some of the limitations of the DMA controller, IBM added an enhanced DMA controller into their PS2 machines. Unfortunately, the BIOS of the PS2 machines does not support these extended capabilities.

Channels and priorities

Since the DMA controller has four channels it can serve 4 different devices. However, only one of these channels can be active at a time, and therefore, a DMA transfer is performed on only one channel at any one time. Due to this limitation, a priority mechanism analogous to the interrupt controller is necessary. This is the only way the DMA controller can determine, which one request will be served first when concurrent requests are pending.

Like the interrupt controller, the DMA controller supports, in terms of priorities, two different operational methods: One with static and the other with dynamic (rotating) priorities. PC's basically work with static priorities where channel 0 represents the highest priority and channel 3 the lowest. This order is established by a chip and cannot be changed. It is up to the board designer, however, to connect each peripheral device at board level to a channel via hardware. Each device has its own connection available to the controller, which we'll talk about in the next section on DMA connections.

When multiple DMA devices simultaneously request a DMA transfer, the device at channel 0 always receives top priority. The device on channel 3 always receives the lowest priority. However, you can also disable individual channels using an internal mask register. In this case, the DMA controller ignores the DMA commands from the respective device.

The DMA controller connections

The original 8237A DMA controller was designed as a 40 pin chip. Today, there are several variations including integrated chip sets. Regardless of changes in the "packaging", the function of the DMA has not changed.

The following table shows the different connections of the DMA controller.

DMA Controller Connections			
Line	Pin	Name	Task
-IOR	1	IO-Read	Depending on the condition of the DMA controller, this signal has different meanings. In the wait state, the CPU activates this line when it reads data from one of the controller's internal registers. During a DMA transfer, the DMA controller activates this line when it transfers data to main memory.
-IOW	2	IO-Write	This signal works analogous to -IOR. It refers, however, to writing procedures. In the wait state, the CPU activates this line when it writes data into one of the internal DMA controller registers. During a DMA transfer, the DMA controller activates this signal when it transfers data to a peripheral device.
-MEMR	3	Memory	The DMA controller activates this line when data from the main memory needs to be read.
-MEMW	4	Memory Write	The DMA controller activates this line when data needs to be written to the main memory.
Voc	5		Voltage supply (depending on the system +3,3V or +5V)
Ready	6		Main memory and peripheral devices activate this line to indicate that they have received data from or are ready to initiate a DMA transfer. This line allows slow peripheral devices or memory a way to lengthen the reading or writing access to the DMA controller.
HLDA	7	Hold Acknowledge	The CPU or another controller activates this line when it has released the local bus. This indicates that the DMA controller can assume control of the bus for transfer.
ADSTB	8	Address Strobe	The DMA controller uses this line to load the most significant byte of a transfer address into the external DMA latch register. It activates this line as soon as it has placed this address on lines DB0 to DB7. 12.5 cm 0 frame
AEN	9	Address Enable	The DMA controller activates this line during a DMA transfer to command the external DMA latch register to write its contents as part of the memory address on the lines A8- A15 of the address bus.
HRQ	10	Hold Request	The DMA controller activates this line to request control of the bus from the CPU or from a bus master. Line HLDA is activated when the request is satisfied.
-CS	11	Chip Select	The CPU activates this line when it wants to read from or write to the DMA controller internal registers. The data is passed on lines DB0-DB7, which then functions as local data bus.
CLK	12	Clock	This is the clock speed for the DMA controller.
Reset	13		The CPU activates this line to return the DMA controller to its initial status.
DACK0-DACK3	14,15,	DMA-Acknowledge	After a DMA request on one of the DREQ lines, the DMA controller informs the peripheral over one of these lines that it is ready to start the DMA transfer. Since only one device can be served at a time, the DMA controller activates this signal only when there is no DMA request of a higher priority. In addition, activating this signal is always preceded by assuming control over the bus using the HRQ and HLDA lines to insure that the DMA controller has unimpeded access to the bus. Since only one channel is active at a time, only one of these lines is set at any one time.
DREQ0-DR	16-19	DMA Request 0-3	Each of these lines is connected to one of a maximum of four peripheral devices, served by the DMA controller. If one of these devices initiates a DMA transfer, it activates the DREQ line and, in doing so, informs the DMA controller of an intended transfer.
GND	20	Ground	

DMA Controller Connections			
Line	Pin	Name	Task
DB0-DB7	21-23, 26-30	Bus 0-7	The CPU accesses the DMA controller registers over these lines which form a two-way data bus. In addition, the controller uses these lines to write the high order of the address (bits 8-15) to the external DMA latch register. power supply (depending on the system +3.3 or +5 Volt)
Vcc	31		
AO-A3	32-35	Address 0-3	Depending on the mode of the DMA controller, these lines have different purposes. In the wait state, during a read or write access of the DMA controller's internal registers, the CPU places the register number on these lines (maximum of 16 registers). During a DMA transfer, the DMA controller writes the least significant four bits of the memory address (bits 0 to 3) to the bus.
-EOP	36	End of process	The DMA controller activates this line to indicate to the device the end of the current DMA transfer. A connected device can also activate this line to terminate a transfer. In this case, the transfer is terminated and the respective flags in the DMA controller's internal registers are reset.
A4-A7	37-40	Address 4-7	During a DMA transfer, the DMA controller passes bits 4-7 of the memory address to the bus over these lines.

The DMA controller registers

The DMA controller has 27 internal registers - an ample supply for a simple device. Certain registers appear four times to accommodate each of the four DMA channels.

DMA Controller Registers					
Register	Number	Width/bit	Register	Number	Width/bit
Starting address	4	16	Status	1	8
Counter	4	16	Command	1	8
Current address	4	16	Intermediate memory	1	8
Current counter	4	16	Mode	4	8
Temporary address	1	16	Mask	1	8
Temporary counter	1	16	Request	1	8

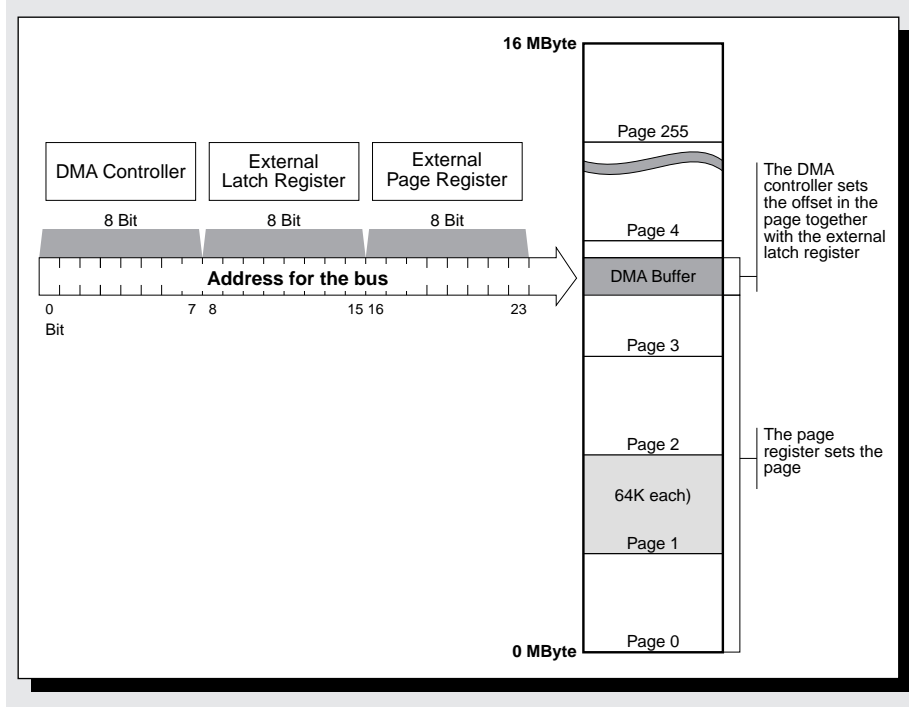
The counter register specifies the number of bytes to be transferred for a channel. A 16-bit register holds a maximum value of 0FFFFh. This corresponds to a maximum transfer of 64K - the number of bytes transferred is one more than the value in the counter. The reason for this is that the DMA controller decrements the contents of the counter register after each byte and continues until the value in the counter changes from 000h to 0FFFFh. Therefore, to transfer a single byte, you load the counter register with the value 000h. The cycling of the counter register from 000h to 0FFFFh has a special name in DMA terminology: "Terminal Count" abbreviated TC. We'll see this term again in the next section.

Similarly, the registers which specify the base address for the DMA transfer on their respective channels are also 16-bits wide. The DMA controller can address only the first 64K of main memory if a channel has not been assigned an additional external DMA page register. With the DMA page register, the controller can address bits beyond A15 and thus provide unlimited access to the entire address space.

On the PC/XT with its 20-bit wide address bus, the DMA page register contains address bits A16 to A19. On an AT with its 24-bit address bus, the DMA page register contains address bits A16 to A23.

To complicate matters even more, the DMA controller specifies only address signals A0 to A7 on its lines directly. Address bits A0-A7 are generated by the DMA controller and address bits A16-A19 (or A16-A23 on an AT bus) from the DMA page register. But note that there's a gap in the address. Address bits A8-A15 are missing.

Address generation through the DMA controller



This gap is filled by the DMA address latch. This is an external 8-bit register connecting the DMA controller to its data bus DB0-DB7 through the directional line ADSTB. The DMA controller writes address bits A8-A15 to this register over line DB0-DB7, which are then placed on the address bus by the DMA address latch as soon as a transfer is initiated.

DMA-segment overflow

Because the DMA controller keeps address bits A0-A15 separated from the address bits A16-A19 (or A16-A23 on an AT), there is always the danger of a DMA segment overflow during a DMA transfer. This happens when the lower address bits A0-A15 cycle from 0FFFFh to 000h, meaning a change from the last byte of the current 64K page to the first byte of the following 64K page is taking place. The DMA controller should increment address bits A0-A15 and also increment the contents of the DMA page register. But this does not happen. Instead, the next byte is transferred to the same 64K page, but at offset 000h.

This example illustrates the overflow: Assume that you allocate a buffer in memory to accept data from a DMA transfer at memory address 0000:E000. This is within the first 64K page of the main memory. The buffer is 16K in length (4000h). Therefore the buffer extends to address 1000:2000. The first 8K of the buffer are within the first 64K page of the memory. The second 8K of the buffer are in the second 64K page.

During the DMA transfer, the DMA controller writes the first 8K of data into the buffer. Immediately thereafter, the problem begins. As the offset changes from 0FFFFh to 000h, the DMA controller "forgets" to increment the DMA page register and begins to transfer the second 8K of data. However, this data isn't transferred into the second 8K of the buffer as expected, but rather winds up at the beginning of the first 64K page. A problem like this will soon cause the system to crash since the DMA controller can overwrite data or program code.

As a programmer, you have to prevent this from happening. If the dividing line between two 64K pages is transgressed, the transfer must be split into two blocks. It's therefore important to first program the DMA controller so it is dedicated to the first part of the buffer below the 64K byte limit. Upon completing this transfer, the DMA controller is again be programmed

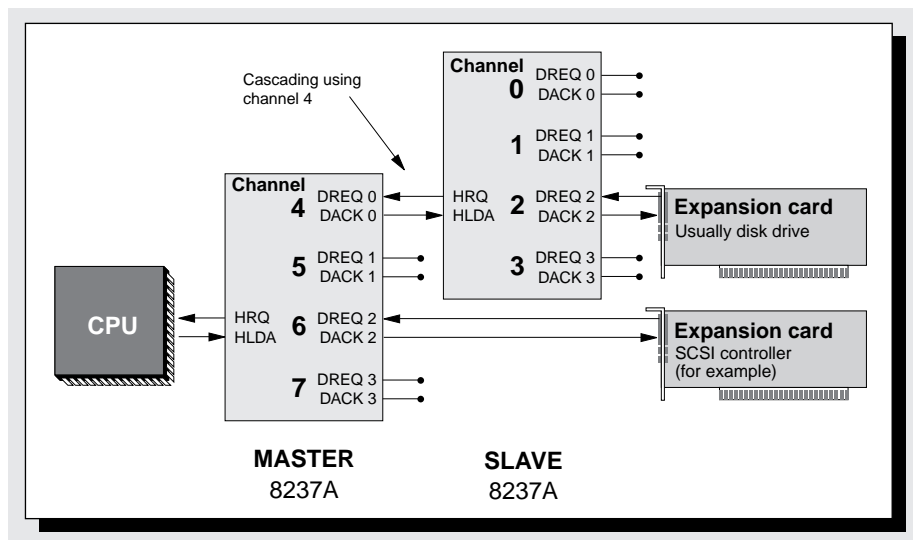
to execute the transfer into, or out of, the second buffer beyond the 64K limit. How the DMA controller is programmed is discussed later in this chapter.

DMA architecture in the PC/XT or AT

If the basic functions of the DMA controller are not easy to understand, programming the DMA controller can be even more difficult. The main reason for this is that we're dealing with different DMA architectures for a PC/XT and an AT. On an PC/XT, a single DMA controller performs only 8-bit transfers. On an AT, a second DMA controller can perform 16-bit transfers.

The two DMA controllers in the AT are cascaded (connected to a larger unit, as is also commonly done with the interrupt controller). The slave is then connected to one of the master channels by connecting the HRQ and HLDA connections with the master's DREQ and DACK connections. To be more precise, in the master these are the connections DREQ0 and DACK0, because with the AT, the slave DMA controller is always connected to the master channel 0. Cascading through channel 0 results in all the slave channels having a higher priority than those of the master. According to this priority ranking, the slave channels on the AT are labeled DMA channels 0 to 3, while the master channels are labeled 4 to 7.

*Cascading two
DMA controllers
on the AT*



Master channels 4 to 7 on the AT are designed for 16-bit transfers. The slave, as with the PC/XT, can perform only 8-bit transfers. The tables below show how each channel is used.

Use/application of the DMA channels with the AT			
Channel	Master /slave	Width	Use/application
0	Slave	8 bit	memory-refresh
1	Slave	8 bit	unused
2	Slave	8 bit	disk drive
3	Slave	8 bit	unused
4	Master	16 bit	cascading the slave
5	Master	16 bit	unused
6	Master	16 bit	unused
7	Master	16 bit	unused

Using the DMA channels in the PC/XT		
Channel	Width	Application/use
0	8 bit	Memory refresh
1	8 bit	Free
2	8 bit	Disk drive
3	8 bit	Hard drive

16-bit transfers

Although you can refer to 16-bit transfers in the master DMA controller in an AT, you should note that controller itself can perform only 8-bit transfers. This is best illustrated, by the fact that for each transfer, the internal address registers are incremented or decremented by one only byte and not two, which happens in 16-bit transfers.

Nevertheless, 16-bit transfers can occur by carefully interconnecting the DMA controller channels with the address and data bus. First, remember the DMA controller never reads the bytes or words that are to be transferred, but only directs the signals on the data, address and directional or control buses so the data are transferred from a peripheral device into the memory (or in the opposite direction). Therefore, when changing from an 8-bit to a 16-bit transfer, the internal register do not have to be expanded.

For 16-bit transfers, the DMA controller then needs only to be interconnected to the control bus master so the connected units recognize, through a signal, that words instead of bytes are to be transferred.

However, one problem remains: The DMA controller increases or decreases the internal address register by only one byte. Fortunately, there is a simple solution for the problem: The DMA controller's address lines are simply shifted one line (or one bit) from the address line of the address bus. The line A0, by which the DMA controller prints out the lower valued address bit, is connected to the line A1 at the address bus, and the same is true for A1 of the DMA controller with A2 of the address bus, etc.

The addresses originating from the DMA controller are all shifted one bit to the left, which is equivalent to multiplying by two. Incrementing the internal address register means the address is incremented by one word. This method "steps through" the data in the area to be transferred one word at a time.

Programming the DMA controller for 16-bit transfers is a little different. First, the desired memory address must be halved when initializing a DMA controller register. The hardware automatically doubles this address. Additionally, the buffer must start at an even memory address. The transfer length is specified as words and not as bytes. To transfer 160 bytes, for example, specify 80 words. Is isn't possible to transfer an odd number of bytes.

Since the maximum value for this 16-bit register remains 0FFFFh, a maximum of 64K words can be transferred (equivalent to 128K). Providing that you start at the beginning of a 64K page in memory, you won't have to be concerned about a DMA segment overflow.

Modes of operation and auto-initialization

The DMA-controller recognizes three different modes of operation which determine the type of transfer:

- Single transfer
- Block transfer
- Demand transfer

What is commonly called a DMA transfer is actually a block transfer. Once triggered by a signal on one of the DREQx lines, the DMA controller executes the data transfer within a block transfer until either the respective counter register jumps from 000h to 0FFFFh (Terminal Count) or the initiating peripheral device stops the transfer with a signal on the -EOP line. During the transfer, the bus remains under the control of the DMA controller, so the CPU cannot access the bus. In certain situations, it's not desirable to isolate the CPU from the bus for the duration of a complete transfer, so the DMA controller also supports single transfers.

For a single transfer, only one byte (or word in 16-bit channels) is transferred before the DMA controller automatically terminates the transfer. Although the counter register is simultaneously decremented and the address register is incremented, a new command has to initiate the next single transfer. In this manner, the CPU has time after each individual transfer to assume control of the bus before the next single transfer is initiated.

Demand transfer is similar to block transfer, but the DREQx-line has a somewhat different purpose here. Although an short signal on the DREQx line is sufficient to initiate a transfer in a normal block transfer, this line is stay active during a demand transfer. When the respective peripheral device reassumes the line, the DMA controller temporarily halts the transfer until the line is reactivated. When this happens, the transfer restarts at the place where it left off.

This mode is important for devices that independently want to indicate the end of a transfer by signalling on -EOP line in a normal block transfer.

Regardless of the mode of operation, the DMA controller can perform write, read and verify transfers. Data are written from a peripheral device to main memory during a read transfer. Data are written from main memory to a peripheral device during a write transfer. During a verify transfer, no data are moved.

An extremely practical feature of the DMA controller is "auto initialization". This is also available in all three modes of operation. With auto initialization on, when Terminal Count is reached or -EOP line is active, the DMA controller will put the address and length of transfer that was last programmed in the respective channel. This saves the programmer from having to continually set the respective registers when a block having a constant length is transferred to fixed location in memory.

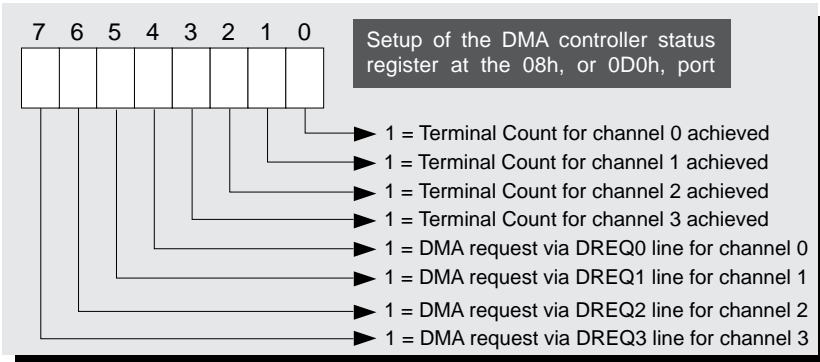
Programming the DMA controller

The following table shows the different DMA controller registers which are used to determine the status of the controller or define the parameters:

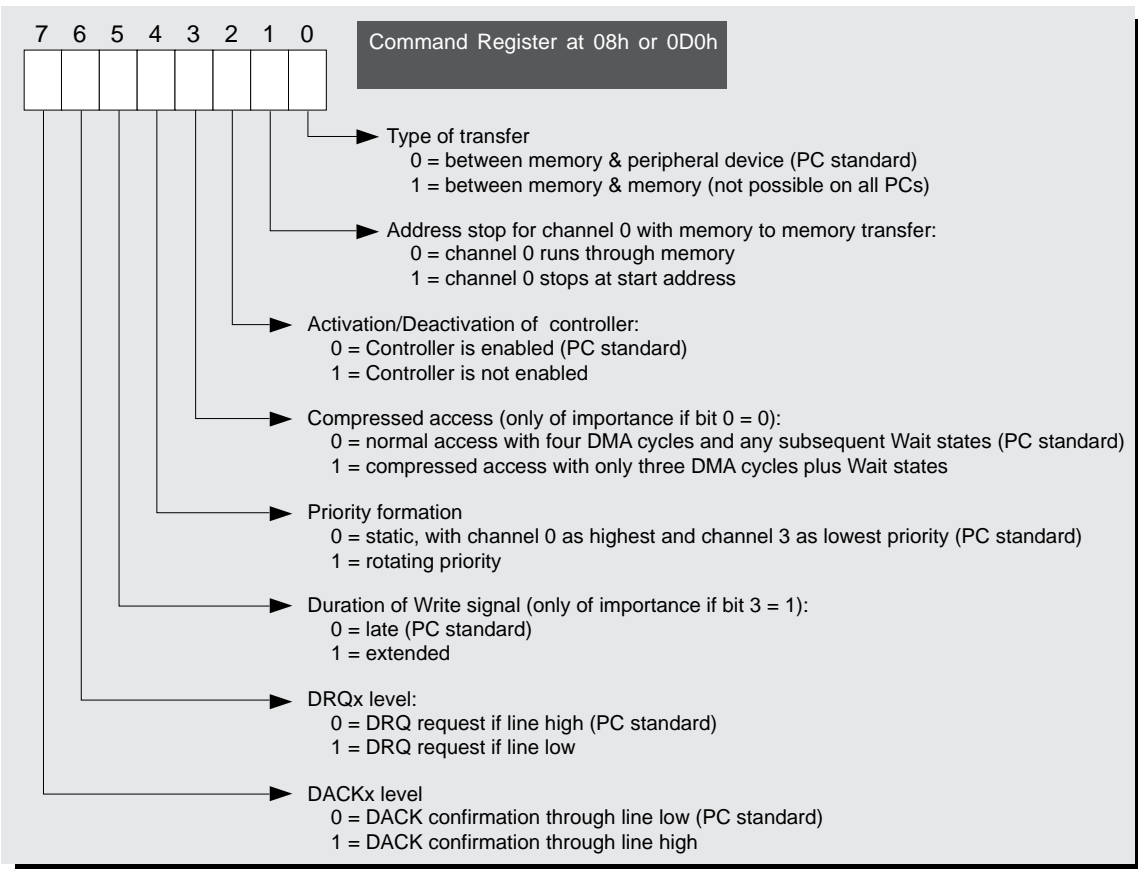
DMA register in the PC/XT (or AT) that directs the DMA controller				
Register	Port*	Port**	Read	Write
Status	08h	ODOh	x	
Command	08h	ODOh		x
Request	09h	OD2h		x
Masking	10Ah	0D4h		x
Mode	0Bh	0D6h		x
ByteWord-FlipFlop	0Ch	0D8h		x
Intermediate memory	0Dh	0DAh	x	
Reset	0Dh	0DAh		x
Masking reset	0Eh	0DCh		x
Masking	0Fh	0DEh		x
* slave in an AT / only one DMA in a PC/XT				
** master in an AT / not present in a PC/XT				

Before you access one of these registers, decide if you're addressing the master or the slave. If you have a PC/XT that has only one DMA controller, it's not possible to access a second DMA controller (master in the AT).

The status register lets you determine if there is a hardware request (a signal on the respective DREQx line), or a Terminal Count for one of the different channels. The latter indicates that a DMA transfer has been completed on that channel. A special feature of this register is to reset the initial values of the four TC flags. By selecting the register, these flags are cleared.



The command register is located at the same port address as the status register. It goes through several settings on the DMA controller. Some of these settings, especially the 5 bit, are interesting in certain situations for DMA programming using software. The problem with this register, however, is that it cannot be selected and used due to the status of the other bits. In regard to the other bits, you must depend on the fact the BIOS has selected the standard setting for the PC when the system was booted and that you can take over these default settings without causing any damage.



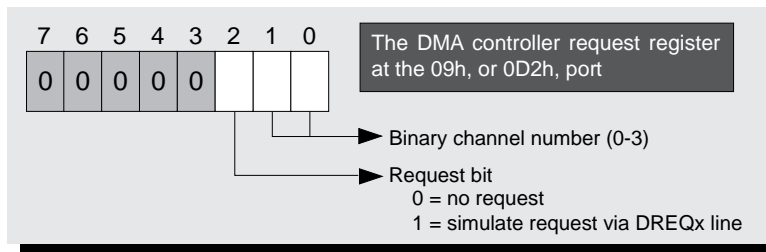
The 7 and 6 bits of the command register determine the polarity of the DREQx and DACK signals. According to the standard PC settings, both bits should be set to 0. The DMA controller recognizes a DMA request in DREQx if this connection is set

to high and the corresponding DACKx line is set to low, providing it follows the DMA request. Bits 5 and 3 determine the signal length for read and write impulses and are not important for programming.

Bit 4 determines the priority selection. In the PC, the DMA controller uses static priorities, where channel 0 is assigned the highest priority and channel 3 the lowest. Therefore, always set this bit to zero.

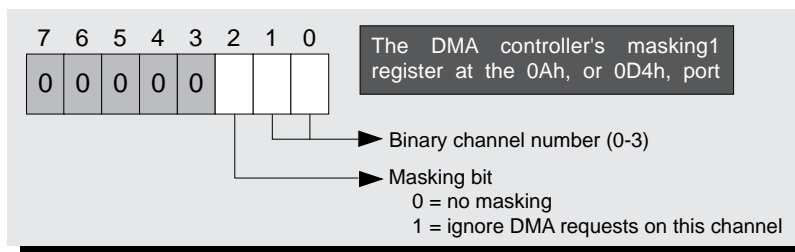
Bit 2 activates and deactivates the DMA controller. For example, you can set the controller to a state in which the CPU can read and write to the registers, but doesn't respond to any DMA requests. A typical use for this is for changing the controller settings and but want to prevent a DMA transfer at the same time. Since channel 0 and the memory refresh are blocked when the DMA controller is deactivated, use this for as brief a time as possible. An alternative is to use the mask register to mask only the channel that you wish to change.

The request register is used to initiate a DMA transfer under software control. This is done by simulating the activation or clearing of one of the DREQx lines. The request register is also used to initiate a memory to memory transfer, since a peripheral device is not involved and therefore cannot send a signal over a DREQx line.



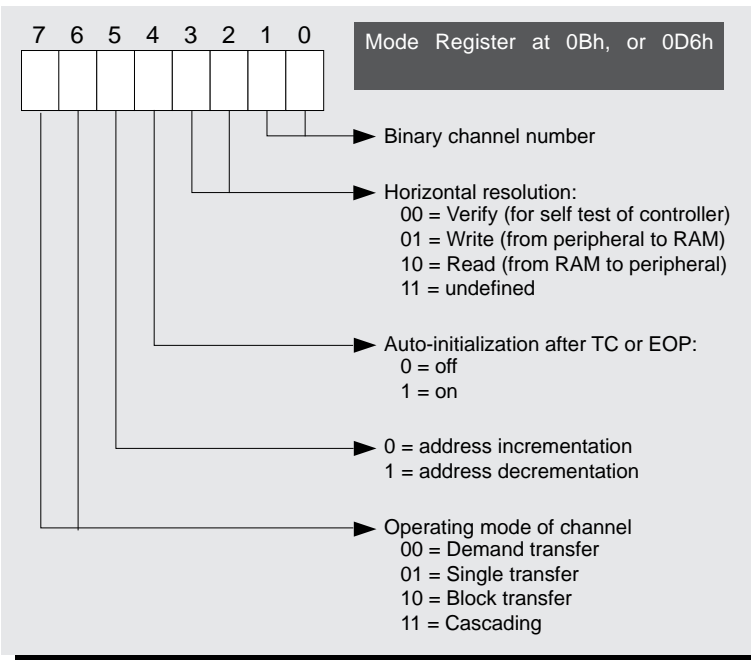
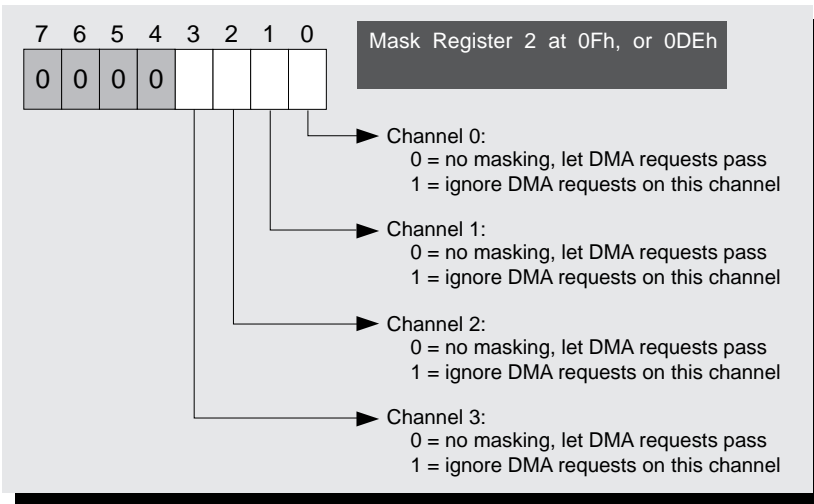
If a DMA request is simulated over the request register, its response depends on whether other, higher priority DMA requests are pending. If so, the DMA request must wait for its turn to be serviced. During this time, the request can again be turned off using the request register. This is done by setting the 2, called the request bit, for the respective channel number.

There are two mask registers. Mask register 1 is organized similarly to the request register. It is used to either turn off or reactivate a channel. If a DMA channel is to be deactivated, the preferred option is to turn off the channel since the channel register (start address, length, etc.) would have to be reset. In such an instance, the consequences would be fatal if a DMA transfer were released on this channel while the register indicates piece by piece its old contents and has already been reinitialized.



Another way to mask or make a channel receptive to DMA requests is provided by mask register 2. In contrast to the mask register 1, all four channels are affected. Use this register only to change the status for all four channels simultaneously.

As its name suggests, the mode register determines a channel's operating mode. You can specify if the next DMA transfer will happen as a single transfer, a block transfer, or a demand transfer. It also specifies if the channel is to cascade two DMA controllers. In most cases you won't have to change this later setting since this happens when the computer is booted.



Bit 5 of the mode register, determine the "direction" of a transfer. This "direction" isn't to or from a peripheral, rather it's forward or backward direction in memory. So you can decrement instead of increment the memory address during a DMA transfer. In this case, a data block is read backwards to forwards by the peripheral. Also the ending address of the buffer is loaded into the proper register before starting the transfer.

The mode register also lets you can auto-initialize a channel after reaching Terminal Count, or after receiving a signal on the -EOP line. The channel number is specified by bits 0 and 1 and the operating mode by bits 2 and 3.

Three of the ports on the DMA controller - 0Dh, 0Eh and 0Fh and the corresponding 0D8, 0DAh and 0DCh - are not essentially don't have a register, but are used to receive specialized commands. When the CPU writes to one of these ports, it releases a particular function in the DMA controller (regardless of the contents of the data that was written). By writing to port 0Eh

(or 0DCh), masking for all channels is removed and the controller again responds to DMA requests on these channels. The same is true for the 0Dh port (or 0DAh); however, the DMA controller is completely reset so the command, status and request registers are returned to their original settings.

The third "command port", 0Ch (or 0D8h), is used to set the 16-bit register with the starting address for a DMA transfer and for the length of the transfer. This is shown in the following table:

DMA register in the PC/XT or AT for setup and query of the DMA channel					
Channel	Register	Port*	Port**	Read	Write
0	Start address	00h	0C0h		x
0	Current address	00h	0C0h	x	
0	Transfer length-1	01h	0C2h		x
0	Remaining length-1	01h	0C2h	x	
1	Start address	02h	0C4h		x
1	Current address	02h	0C4h	x	
1	Transfer length-1	03h	0C6h		x
1	Remaining length-1	03h	0C6h	x	
2	Start address	04h	0C8h		x
2	Current address	04h	0C8h	x	
2	Transfer length-1	05h	0CAh	x	
2	Remaining length-1	05h	0CAh	x	
3	Start address	06h	0CCh		x
3	Current address	06h	0CCh	x	
3	Transfer length-1	07h	0CEh		x
3	Remaining length-1	07h	0CEh	x	
*Slave in an AT / only one DMA in a PC/XT					
**Master in an AT / not present in a PC/XT					

To set up one of these registers to determine the start address or the length of a DMA transfer, you must output to port 0Ch or 0D8h. An internal FlipFlop, lowered to zero, shows the state of a 16-bit transfer. After the FlipFlop is lowered to zero, it sends the low-order byte of the address to the port, for example port 0C4h for channel 1 of the AT master DMA controller (AT channel 5). This output trips the internal FlipFlop. The port now knows that the most significant byte of the address is coming. This procedure is necessary because access to the different 16-bit registers has to fit into the 8-bit wide DMA hardware. Therefore, a 16-bit value has to be divided into a low byte and a high byte. And since the low and high bytes are transferred at the same port, the DMA controller needs the FlipFlop to differentiate between the two.

Selecting a 16-bit register proceeds in the same manner: First, the FlipFlop is lowered. Next the port is read to get the low-order byte. Finally, read the port to get the high-order significant byte.

After the mode register, the start address and the transfer length, one other task remains. So as not to be limited to the first 64K of memory for the transfer buffer (128K for a 16-bit channel), the DMA page register for the respective channel must also be set. The following table shows the position of the DMA page register which accepts the address bits 16-23 of the transfer area. Remember, the page register for channels 4 to 7 are only available for AT's having a second DMA controller.

The table on the right lists the DMA page register for address bits 16-23 of the transfer area.

Channel	Port	Channel	Port
0	87h	4	8Fh
1	83h	5	8Bh
2	82h	6	89h
3	81h	7	8Ah

A special feature is found in the PC/XT page register. For channels 0 and 1, there is only one page register available, but it can be addressed over both of the port addresses.

DMA utility

The module DMAUTIL is an example of programming the DMA controller. You'll find both the listing and the program on the companion CD-ROM. DMAUTIL contains all the important functions that you'll need to set up channel parameters and to initiate a DMA transfer. You'll find an example program with the DMA utility functions in Chapter 38 on how to program Sound Blaster or compatible cards.

You'll find the following program(s) on the companion CD-ROM



DMAUTIL.C (C listing)
DMAUTIL.H (C listing)

DMAUTIL.C Functions	
Function	Task
dma_Masterclear	Initiate DMA controller reset
dma_SetRequest	Initiate DMA transfer on the specified channel
dma_ClrRequest	Halt DMA transfer on the specified channel
dma_SetMask	Mask (block) specified channel
dma_ClrMask	Clear specified channel
dma_ReadStatus	Read DMA controller status
dma_ClrFlipFlop	Clear FlipFlop for 16 bit register access
dma_ReadCount	Read channel transfer counter
dma_SetChannel	Set DMA channel for transfer
dma_Until64kPage	Determine the number of bytes within the buffer, which lie on a 64K-Page
dma_AllocMem	Allocate DMA capable memory
dma_FreeMem	Free DMA capable memory



Serial Ports

Like the parallel port (see Chapter 8), the serial port is basic to the PC. Every PC, even subnotebook computers, have at least one serial port. Today's PCs usually have two serial ports: One with a small connector and another with a large connector. Serial ports are most commonly used to attach a modem or a mouse to the computer but they can also communicate with more exotic devices such as clock radios, model trains and other devices.

A serial port is much slower than a parallel port. Instead of eight bits at a time, the serial port sends each bit through a line individually. Given the same clock speed, the serial port is eight times slower than the parallel port. What is responsible for this speed disadvantage however is precisely what gives the serial port its superior cabling properties. This port operates "serially" - instead of eight data lines plus control lines, the serial port consists of just one line for grounding and one for data (data transmission in both directions at the same time requires two data lines, but this is still a total of only three lines). This is why parallel port cables are heavier and more expensive.

When devices are in close proximity, the cable cost is not a significant factor. When devices are distant, cable costs are a much more important factor.

With only two lines (in its simplest form), the serial port can do some remarkable things. Telephone cables are potential serial transmission cables for a PC. To connect a PC in New York, Buenos Aires or London with a PC in Bombay, you don't have to run a parallel cable halfway around the world - you simply dial the computer in Bombay. Serial ports have been used since the 1960's to connect terminals and printers with their host computers, using the existing telephone cables already in office buildings.

The serial port remains today the best way to link remote devices with a PC. Connecting a modem or a mouse to the serial port also has historic origins. Early IBM/PC computers included both a parallel and a serial port, even though in most cases only the parallel one was used (for the printer). When modems and mice arrived, the obvious choice for them was the serial port - this way the user didn't have to invest an additional interface cards.

While a new standard has already become necessary for the parallel interface (see Chapter 8), the standard for the serial interface has remained the same since 1969. It was at this time that the EIA (Electronic Industries Association) established the RS-232C standard. This standard defines the data transfer protocol, cables, electrical signals, and connectors for an RS-232 connection. So when you see the characters "RS-232" appearing on a device, you're able to use it with your PC. Your PC is guaranteed to have an RS-232 interface, although it is usually called the "serial" interface.

As one user types characters on a terminal (or PC with a terminal program), they appear almost immediately on the screen of another user hundreds of miles away. This interaction requires more than just a serial port. You also need:

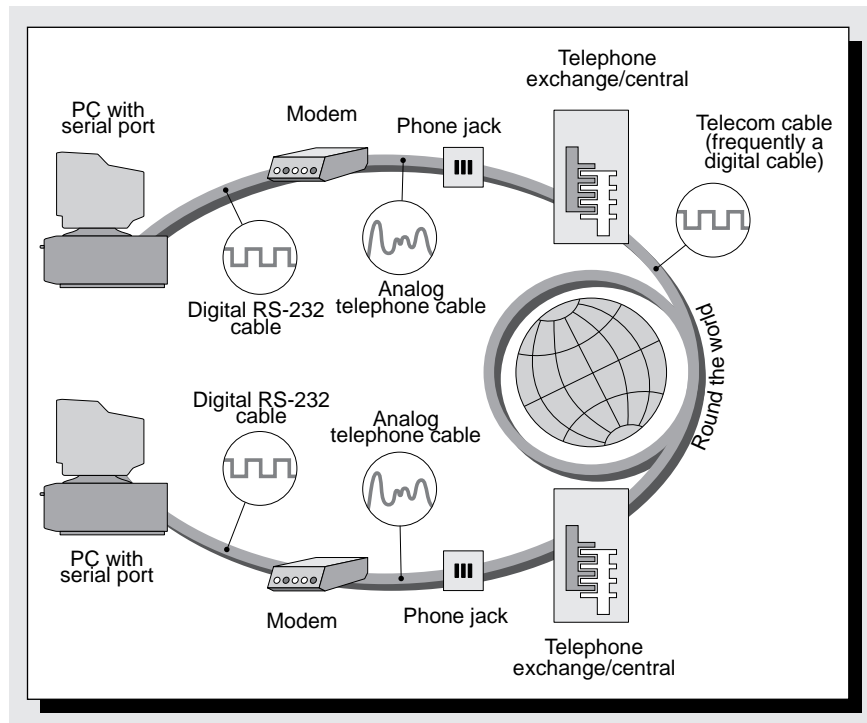
- The terminal program to instruct the serial interface controller to send and receive characters.
- The controller to convert a character to be sent into bit sequences, sends them over the appropriate lines and in the opposite direction, interprets incoming signals, converts them to bits and combines them into bytes.
- The modem to receive characters through the serial interface and sends them over the telephone network in the form of acoustical signals. At the same it converts incoming signals back to bits and transmits them to the serial interface.
- The connector and cable to link one serial port with another serial port or modem.

The following sections describe in detail the steps needed for data to travel from one PC to another. Before we continue however, we'll briefly talk about modems.

Although, telephone lines can theoretically connect two serial ports directly, the technology used by the phone system presents a certain obstacle. The analog telephone network, which still prevails today, is not designed to transmit voltage signals. First +12 volts for a logical zero and then -12 volts for a logical one - such signals are not suited to the telephone network. The telephone system demands the transmission of tones, and this is where the modem comes in.

The modem converts signal sequences from the serial port to acoustical signals of defined frequencies, and sends them through the telephone line. The modem at the other end receives these "tones" and converts them back to the signal sequences expected by the serial port. In this way two serial ports appear to be directly connected. Modulators and demodulators within the modem are responsible for converting (modulation) of electric signals to tones and tones to signals. This is how a modem gets its name: From mod(ulation) and dem(odulation).

*PCs connected
through a
telephone
network*



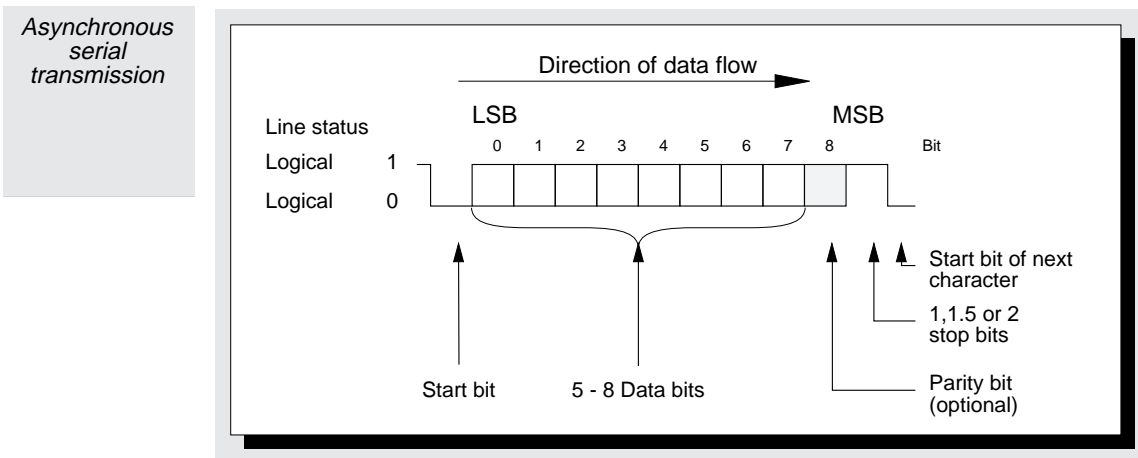
Synchronous And Asynchronous Communication

The two types of communication we'll talk about are *synchronous* and *asynchronous* communication. Synchronous communication occurs when sender and receiver share a common signal pulse. This common signal pulse helps to synchronize their actions. For example, the sender always sends the next bit over the line immediately prior to a new clock pulse and the receiver knows that when the clock pulse occurs it can now retrieve this bit. Sender and receiver are therefore "synchronized".

Synchronization always requires an extra line. The sender and receiver exchange the signal pulse through this extra line. Such a line, however, does not exist in serial transmission through a two-line cable. Both available lines are used one for grounding and the other for data.

Therefore, synchronization for serial transmission must occur on the data line through the data itself. A "data word" to be sent will have status information placed before it and after it, in accordance with the RS-232 protocol. The data words, which can consist of 5-8 bits, are therefore encapsulated. The exact number of bits is set by the two sides upon establishing the connection, depending on how their serial ports are programmed. The two sides must agree about the number of bits, otherwise the communication will fail.

The following illustration shows that the only applicable line states are 0 (low) and 1 (high) in a serial transmission. When no character is being sent, the line is high, or in "marking condition". When the line state changes to low, it marks the beginning of the data transmission. Depending on the agreement, between five and eight data bits are now be sent over the line. During the transmission, when the line goes low a 0-bit is sent, when it goes high a 1-bit is sent. The lowest-order bit of a character always goes first and the highest-order bit last.



Besides the data bits, a parity bit detects transmission errors. The two types of parity are even parity and odd parity. The parity bit in even parity supplements the data word being transmitted so the number of set bits (those with a value of 1) is even. For example, if the data word contains three bits with the value 1, the parity bit will also be 1. The extra 1 from the parity bit raises the number of 1-bits to four, thereby producing an "even parity". On the other hand, if the word contained an even number of 1-bits, the parity bit would be 0. The opposite occurs with odd parity - the value of the parity bit is such that the total number of 1-bits is odd.

Although parity recognition is well-designed, it often fails because errors are found only when one data bit or an odd number of data bits are "lost" in the transmission, arriving as a 0 instead of a 1 (or vice versa). When an even number of data bits disappears, the parity check still works out despite the underlying transmission error.

In mailbox communication therefore, automatic insertion of parity bits is bypassed. This is easily done by programming the serial interface controller accordingly. Instead, by using checksums, high-level protocols exist which detect errors in the transmission of larger data blocks.

Unlike the parity bit, the stop bit is not optional. A stop bit signals the end of transmission of a data word. The communication protocol allows 1, 1.5 or 2 stop bits (depending on word length). The option of 1.5 stop bits may at first seem strange, but it does bring up a question not yet answered. How does the receiver know when it should interpret the current line state as a data bit? To answer this we need to return to synchronous transmission. For example, if the receiver is reading a bit at the same time the sender sends the next signal down the line, whether the bit arrives correctly is strictly a matter of chance.

The solution is in the transmission speed, which must be identical for both sender and receiver. This transmission speed is measured in baud, or "bits per second". A typical rate of 9600 baud means 9600 bits in one second. The "length" of a bit is, therefore, 1/9600th of a second. The receiver reads the current status of the data line every 1/9600 second and converts it - depending on whether it is high or low - to a 1-bit or a 0-bit.

However, this is not enough to synchronize the sender with the receiver. As in the above case, the receiver can still read the line at the same time the sender places the next bit on the line. This is where the start bit is used. Upon detecting the start bit, the receiver knows the transmission is starting and it must now read the line at fixed time intervals (according to the baud rate). To avoid eventual disruption due to small differences in "time counting", the sender and receiver are resynchronized with the transmission of each data word, through the initial start bit.

We can now answer the question of how a stop bit can be 1.5 bits long. The line is simply set to high or low for 1.5 times the normal "bit-transfer time".

The stop bit ends the transmission of a character. The line now remains in "marking condition" (high) until a new character is sent and the line transmitting the start bit changes to low.

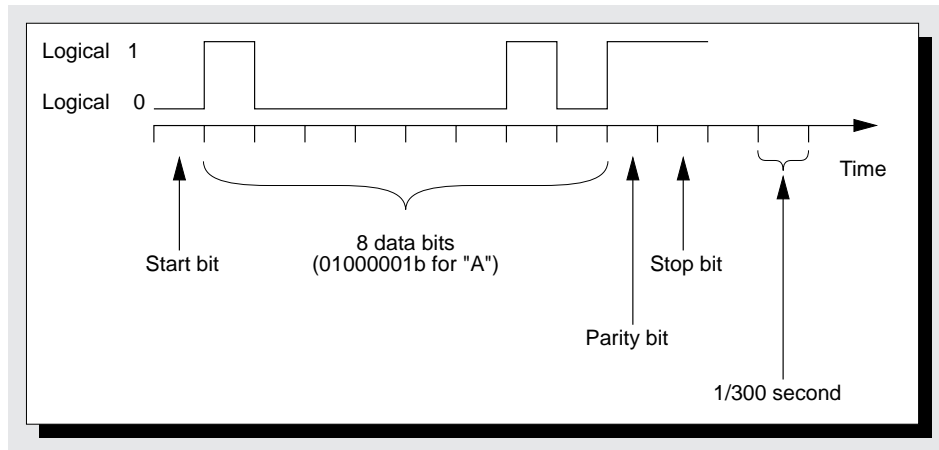
There are also interfaces that work with negative logic. In this case you would switch the 0's and 1's in the above illustration (which corresponds to positive logic). The basic principles of serial transmission remain the same.

Data transmission is successful only if the various protocol parameters are the same for both sender and receiver. The first item that must be set is the baud rate, or number of bits per second. This value ranges from 75 baud to 144,000 baud, the highest transmission rate supported by a PC interface.

The number of data bits transmitted each time depends on the data being sent. Seven data bits are enough for normal ASCII data since the ASCII character set contains only 128 characters. However, eight-bit data words are required for both the full PC character set of 256 characters and binary data. You can also define whether a parity check should occur, and if so, whether parity is odd or even. Both processes yield the same (un)certainties.

Finally, the number of stop bits must be defined. With one stop bit the next character can be sent faster than with two stop bits. Therefore, this setting is very rare in practice. We'll explain later how you can set these parameters either through the BIOS or through direct programming of the serial interface controller.

Transmitting the character "A" with eight data bits, one stop bit, odd parity and 300 baud



The illustration shows the character "A" with a protocol of eight data bits, odd parity and one stop bit. We're assuming positive interface logic and a transmission rate of 300 baud. Since the ASCII code of the letter A is 65 (01000001b) and contains only two 1-bits, the parity bit in this case is 1 in order to produce an odd number of 1-bits.

RS-232 Standard

Upon leaving the serial port, the bits enter the domain of the RS-232 standard. This standard governs the physical dimensions of the connectors, the number and configuration of ports and several electrical parameters. More specifically, we're talking about the RS-232C interface on the PC. Several versions of the RS-232 were designed by the EIA committee until the "C" version finally became official.

The EIA, however, was not the only organization to adopt the RS-232 standard. Another was the CCITT (Comité Consultatif International Télégraphique et Téléphonique), which divided it into two separate standards. The first was V.24 which specified the asynchronous transfer protocol, allocation of individual ports and connector formats. The other standard is V.28 which handles the remaining RS-232 specifications. These specifications include the electrical signal values such as signal level, terminating resistor and circuit stability. The signal level can range from +3 volts to +15 volts for a logical zero and -3 volts to -15 volts for a logical one.

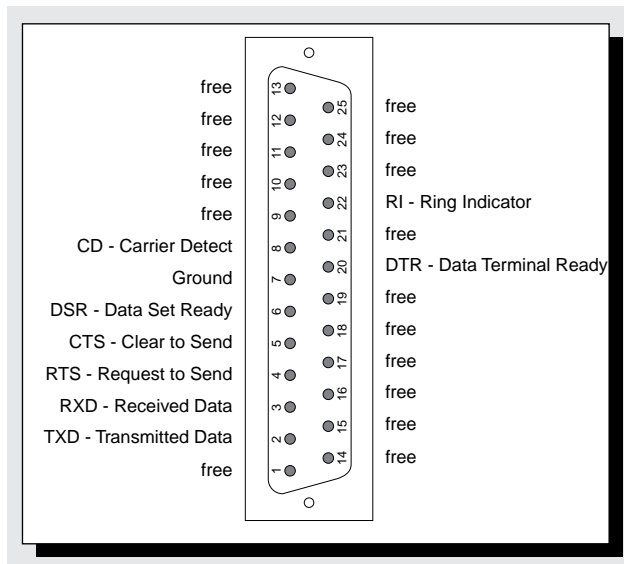
Since RS-232 was intended as a comprehensive standard, there is no mention in the specifications of a particular computer type. Instead they refer to the two poles of the "Data Terminal Equipment" (DTE) and the "Data Communication Equipment" (DCE). In the PC/modem environment, the PC represents the DTE and the modem the DCE. These two devices are connected by an RS-232 cable, as it is commonly known in the industry. The cable becomes unnecessary of course if the PC has a built-in modem card. In this case the card itself functions as the serial port, which is directly connected with the modulator/demodulator (the essence of the modem).

RS-232 connectors and ports

Serial transmission in one direction requires just two lines (grounding and data), while simultaneous transmission in two directions requires just three (an extra data line for the opposite direction). In addition to these, the EIA and CCITT have introduced 17 more lines, which are used for checking and control functions. Luckily not all of these lines need to be implemented for an RS-232 serial data transfer, otherwise an important advantage over parallel cables would be lost.

The following illustration shows the PC version and the wiring inside the 25-pin Submin-D connector. This is a male connector on the PC which connects to the cable leading to the modem.

Implementation of RS-232 lines on the PC, including 25-pin Submin-D connector pin assignments



One of the reasons this connector is so large are the six additional lines through which the DTE (PC) and the DCE (modem) communicate. RTS, CTS, DSR and DTR basically involve "handshaking" between PC and modem when they are communicating in both directions (full-duplex mode), as is usually the case. On many modems the state of several of these control lines can be monitored on the front panel of the modem. The state of the control lines can also appear in terminal programs which you'll then see on the screen. These give you the current status of the connection. Frequently, however, the LED designations do not match the line names given by the RS-232 standard. Instead of RLSD for example, you will often see CD.

Control line definitions for RS-232 cables		
Line	Abbreviation	Definition
Transmitted Data	TxD	The line over which data travels. The DTE (PC) can send data only when all four control lines RTS, CTS, DSR and DTR are logical 1. This line is in "marking condition" (logical 1) when no data is being transmitted according to V.24 protocol.
Received Data	RxD	Data line from DCE (modem) to DTE (PC).
Request To Send	RTS	By setting this line to logical 1, the DTE (PC) asks the DCE (modem) if it is ready to receive data.
Clear To Send	CTS	The DCE (modem) sets this line to logical 1 following an RTS when it's ready to receive the data.
Data Set Ready	DSR	By setting this line to logical 1, the DCE (modem) indicates to the DTE (PC) that a connection has been established with the other side (dialed successfully) and that data can now be sent to the remote DCE (another modem).
Date Terminal Ready	DTR	The DTE (PC) sets this line to logical 1 when it is ready for communication with the DCE (modem). The modem thereby recognizes that it is connected to an active DCE.
Ring Indicator	RI	Through this line the DCE (modem) indicates to the DTE (PC) that there is a call on the telephone line to which the modem is connected.
Received Line Signal Detector	RLSD	Through RLSD the DCE (modem) indicates to the DTE (PC) that it has received a carrier signal from the other end of the phone line. RLSD is also called "Carrier Detect". This does not imply an actual connection however, because the two DCEs (modems) may be unable to find a common transfer protocol for the modulation/demodulation.

Modem communication protocol

Before the PC can send data to the modem, it must set various signals and wait for the desired response from the modem. First, the PC sets the DTR line to high. This tells the modem that the DTE (PC) is ready to communicate with it. The modem responds by setting the DSR line to high. This shows it's ready to communicate. Both lines must remain high during the entire communication. If the DTR line falls back to low, for example if the PC is turned off, the modem resets its DSR line. It then accepts new data only when DTR again goes high. The switching off of DTR by the PC is basically a signal the line should be interrupted. It's used by the telecommunications software on the PC solely for this purpose.

A line refers not only to the connection between modem and PC, but also to the telephone line which the modem opens by dialing. So, if you turn off your PC during a connection but the modem remains switched on, you won't be charged by the telephone company for an open line.

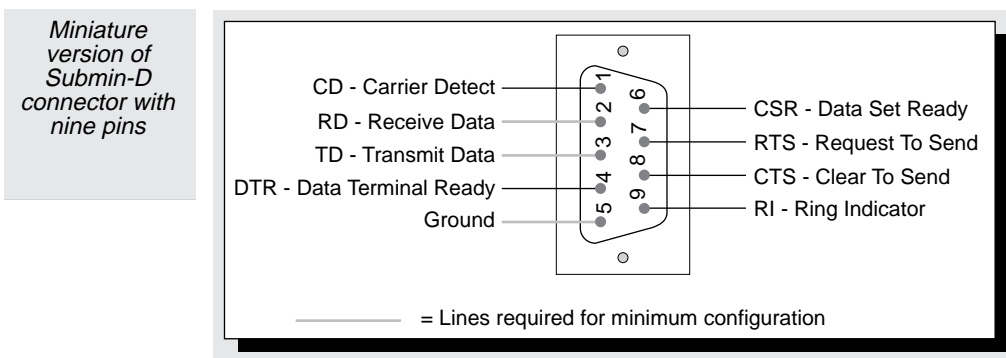
Once DTR and DSR are set, it's still a long way before the computer can actually begin sending characters to the modem. Two handshaking lines are also involved here: RTS and CTS. The PC is the device that first sets the RTS line to high, to signal its readiness to send data. When the modem is ready to receive, it signals back by setting the CTS line. The communications devices are, in a sense, "extending their hands". Once the handshaking is complete, a byte can now be sent. If the PC wishes to send additional data, it simply leaves the RTS line on high and observes the CTS line. Additional bytes can be sent as long as the CTS line also remains high. If it goes back to low, the PC must wait until the modem again signals its readiness to receive by setting the line.

In this process the PC "pumps" data into the modem faster than the modem can output it over the telephone line. The modem collects the data in an internal buffer until it is full. When no more data can be accepted, the modem resets the CTS line until enough data has left the buffer and there is room for more. However if the PC resets RTS because all desired data has been sent, the modem CTS will immediately go "low".

Unlike sending, there is no handshaking for receiving characters. When the PC sets the DTR line, the modem recognizes it as ready and begins sending, whether it is receiving the data or generating it itself. This is also why the above is called a "sender-oriented protocol".

RI and RLSD are two lines which are unrelated to handshaking. RI is required when the PC wants to accept incoming calls. The modem uses the RI line to indicate the telephone is ringing (RI = ring); the PC can then instruct the modem to answer, thereby opening the line.

A more familiar term for RLSD is "Carrier Detect". The carrier is the tone you hear when a modem answers at the other end of the line. The reception of this carrier indicates that although a modem is present, communication may not necessarily occur. Remember, not all modems are compatible due to differences in baud rates or transfer protocols or both. When RLSD is set, however, you at least know you have connected with the other modem.



Without additional changes the above signal lines represent the nine-pin stripped version of the Submin-D connector. This is equally common on PC's as the large version, and contains all the lines necessary to connect a modem with a PC.

Inside The Serial Port

The serial port contains a special chip responsible for input and output of characters, as well as conversion of data words to their corresponding serial interface signals. This chip is called UART (Universal Asynchronous Receiver Transmitter). Another common name for the UART is SIO or "Serial Input/Output".

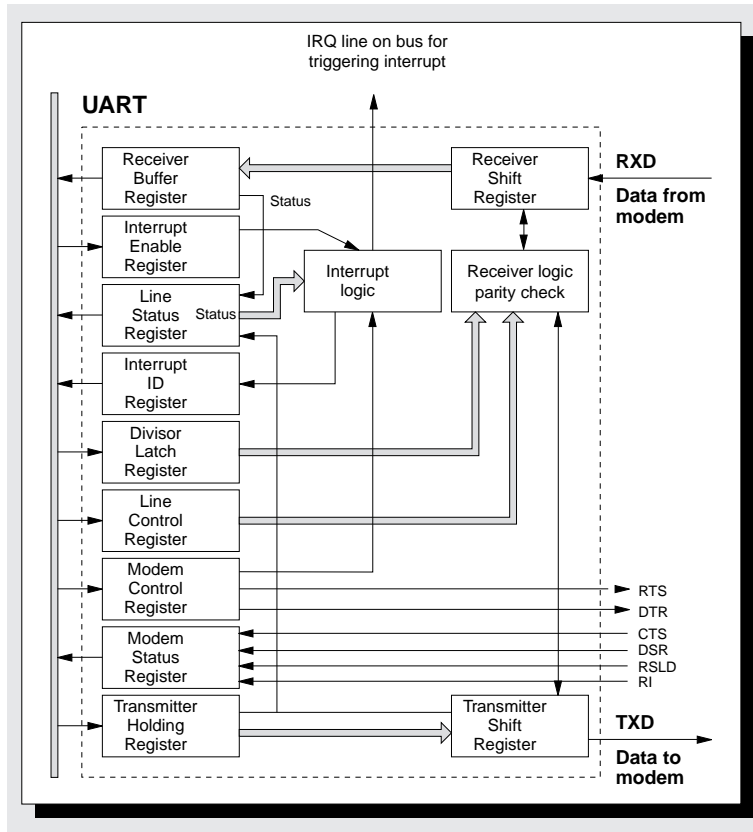
The chip is marked 8250, reminiscent of the Intel DMA and Interrupt Controller. Actually however, it was originally made by the National Semiconductor Corporation (NSC). In the meantime, pin-compatible and function-compatible chips are being offered by a number of other manufacturers such as Siemens and UMC. Today's PCs have an easily expanded and improved version called 16550, which we'll discuss later in this section. Now we'll concentrate on the structure, function and registers of the UART chip.

UART registers

The UART chip contains ten registers that are accessible externally (through software). A few additional registers, such as the receiver shift register and the transmitter shift register, can only be accessed internally. These registers are crucial for sending and receiving characters. When the UART receives a character, the incoming bits first "pile up" in the receiver shift register until a complete data word has arrived. If no error has occurred, the byte is then routed to the receiver data register, where it can be read by the software.

In the opposite direction, the software first writes the data word to be sent to the transmitter holding register. From there the UART moves it into the internal transmitter shift register, from which the individual bits can be sent sequentially down the line. Therefore, the transmitter holding and receiver data registers are the most important in UART data transmission. However, we cannot ignore the other registers because they're needed to initialize the UART and to check the transmission status and line status.

Block diagram
of the UART
chip



Internal UART 8250 Registers					
Register	Abbreviation	Read	Write	Base port +	Bit 7 in Line Control Register
Transmitter Holding	THR			0	0
Receiver Data	RBR	x		0	0
Baud Rate Divisor LSB	DLL			0	1
Baud Rate Divisor MSB	DLM	x	x	1	1
Interrupt Enable	IER			1	0
Interrupt ID	IIR	x		2	-
Line Control	LCR			3	-
Modem Control	MCR	x	x	4	-
Line Status	LSR			5	-
Modem Status	MSR	x	x	6	-

Polling or interrupt

Communication between software and UART can occur either in polling mode or interrupt mode. In polling mode the software is responsible for checking the UART status at regular intervals through the line control register. This is the only way it can tell whether a new character was received or whether the last character sent was actually transmitted. The

corresponding program loops are very easy to write, although they share the disadvantage common to all polling processes: The CPU is occupied with the device the entire time, although relative to CPU speed, characters arrive and depart very slowly.

Depending on your task therefore, you might decide to work in interrupt mode. Although more difficult to program, interrupt processing has several advantages. The CPU devotes itself to the serial port only when a character has actually been received or sent, or when an error has occurred. Only then does the UART initiate an interrupt and activate the interrupt handler, which then deals with the situation. The UART supports this mode through two interrupt registers which we'll discuss later.

Access to registers

The UART registers can be accessed through different ports according to the base address of the serial port. Theoretically, this base address can be freely chosen, but in practice the first two serial ports on a PC (COM1 and COM2) generally use base addresses 3F8h and 2F8h respectively. COM3 and COM4 - if present - normally use base addresses 3E8h and 2E8h.

Base addresses of serial ports and BIOS variables in which they are recorded		
Port	Address in BIOS Variable	Standard Port
COM1	0040:0000	3F8h-3FFh
COM2	0040:0002	2F8h-2FFh
COM3	0040:0004	3E8h-3EFh
COM4	0040:0006	2E8h-2EFh

To ensure that for some reason you "miss" a serial port, you should not use fixed port addresses in your programs. It's better to query one of the four BIOS variables, in which the base addresses of the four ports (maximum supported by the BIOS) are recorded. This way you always get the current address of the desired port.

In reviewing the table with the various UART registers, you can see that two port addresses (base port +0 and base port +1) are reserved by several registers. To differentiate among the various registers when accessing the corresponding port, use

the high-order bit in the line control register. Before accessing any register using the first or second port of the serial interface, load this bit with the value given in the table.

Initializing the UART

To establish a connection with another serial port, the UART must first be initialized. Specifically, you must set the various communication parameters, i.e., baud rate, data word length and number of stop bits. It's also good practice to perform a read access on the receiver data register. Otherwise if a previous program has left a character there, it may be misinterpreted as the first transmitted character of the new connection. On the other hand if there are no characters there, the read won't have any effect. You should not have any problems in either case.

You should start by setting the baud rate because when the UART writes to the corresponding registers, it will reset the other communication parameters in the line control register. So, if you set the register prior to setting the baud rate, you will be forced to repeat the process.

The two registers for setting the baud rate are DLL and DLM. The desired baud rate is not entered directly, however, but is expressed as a quotient with respect to the clock frequency of the UART. This is equal to 1.8432 MHz. The baud rate tells the UART how fast it should generate individual bits in relation to its clock frequency. The following is the actual formula:

$$\text{register value} = 1.8432 \text{ MHz} / (16 * \text{baud rate})$$

The equation shows the UART multiplies the clock pulse duration by 16, and after N of these pulses (N = baud rate) sends the next bit through the line.

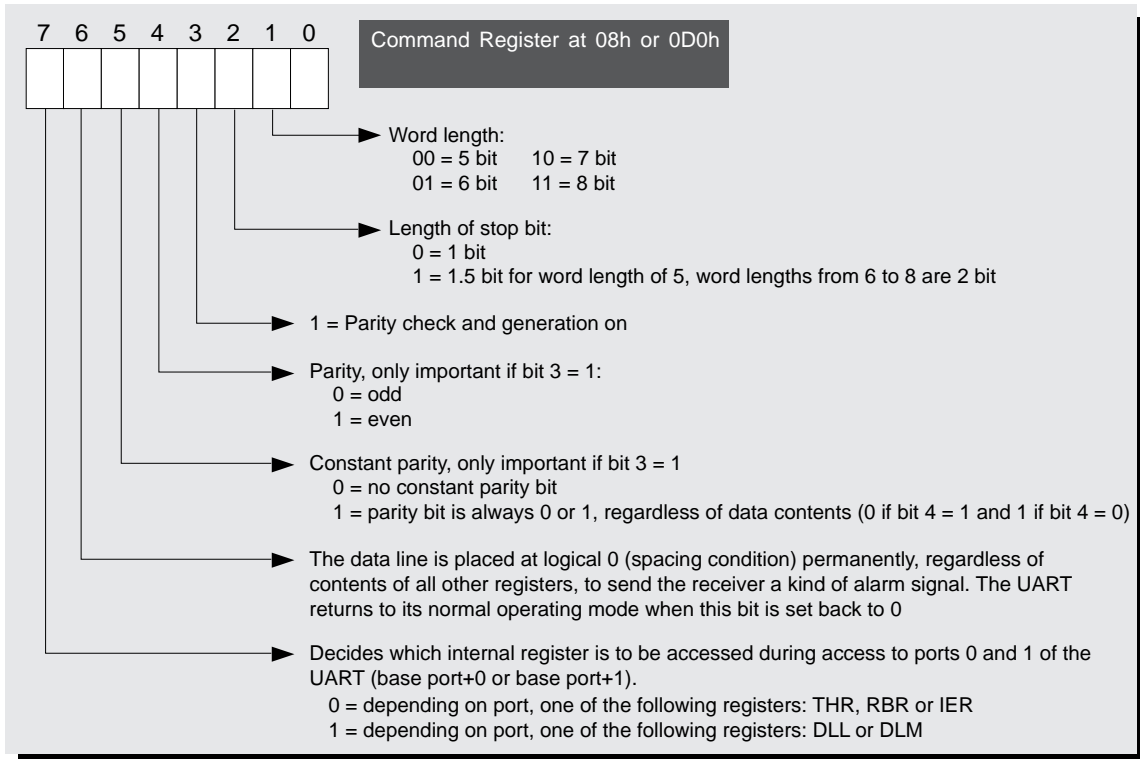
This formula produces a 16-bit register value, whose low-order byte is written to the DLL register and whose high-order byte is written to the DLM register. Theoretically you can set any baud rate, from 1.75 baud (register value = 0FFFFh) to 115200 baud (register value = 1). In practice however only certain baud rates are used, as listed in the following table. The table also lists their corresponding values in the DLL and DLM registers.

Since the highest transmission rate is 115,000 bits per second (115200 baud), and for each 8-bit data word at least two additional bits must be transferred (1 start bit, 1 stop bit), the maximum throughput is approximately 1.4K per second.

However this transfer rate is possible only with direct linkage of two computers and only then with the more modern ones. A 16 MHz 386SX would have a hard time achieving such a high rate - eventually the computer would fall behind.

Common baud rates and their divisor settings in the UART DLM and DLL registers							
Baud rate	Divisor	DLM Reg	DLL Reg	Baud rate	Divisor	DLM Reg	DLL Reg
50	2304	09h	00	2000	58	00h	3Ah
75	1536	06h	00h	2400	48	00h	30h
110	1047	04h	17h	4800	24	00h	18h
134.5	857	03h	59h	7200	16	00h	10h
150	768	03h	00h	9600	12	00h	0Ch
300	384	01h	80h	19200	6	00h	06h
600	192	00h	C0h	38400	3	00h	03h
1200	96	00h	60h	57600	2	00h	02h
1800	64	00h	40h	115200	1	00h	01h

All other parameters such as word length, number of stop bits and use of parity bits are set through the line control register (LCR), located at address 3 relative to the base address of the serial port. Not only can you write to this register to enter new settings, you can also read from it to obtain current settings.



Both bits 5 and 6 have some interesting features. Bit 5 lets you generate a constant parity bit, regardless of the number of 1's in the current data word. This option is not used in practice, however, since it's increasingly being replaced with higher-level software protocols.

Bit 6 allows you to send an alarm to the opposite end. Some programs use this option when sender and receiver need to resynchronize on the software protocol level or when a large data block transmission must be interrupted for some reason. The line is then permanently (until bit 6 is again cleared) set to logical 0. This corresponds to a "spacing condition", which is the exact opposite of the "marking condition" (logical 1) which would otherwise occur when no characters are being sent. The UART at the other end recognizes this and sets a corresponding flag in one of its status registers. Upon checking this register the receiver then becomes aware of the alarm.

It's therefore possible to send information (the alarm information) even when transmission of individual characters or data blocks seems to have gone off track or must be spontaneously interrupted.

Finally, bit 7 is the bit (already mentioned) required to access the registers that share addresses 0 and 1 relative to the base port on the card. To avoid changing the communication parameters in the other bits, you should first read the contents of this register and set or clear bit 7 as needed. The byte obtained in this manner can then be written back to the register.

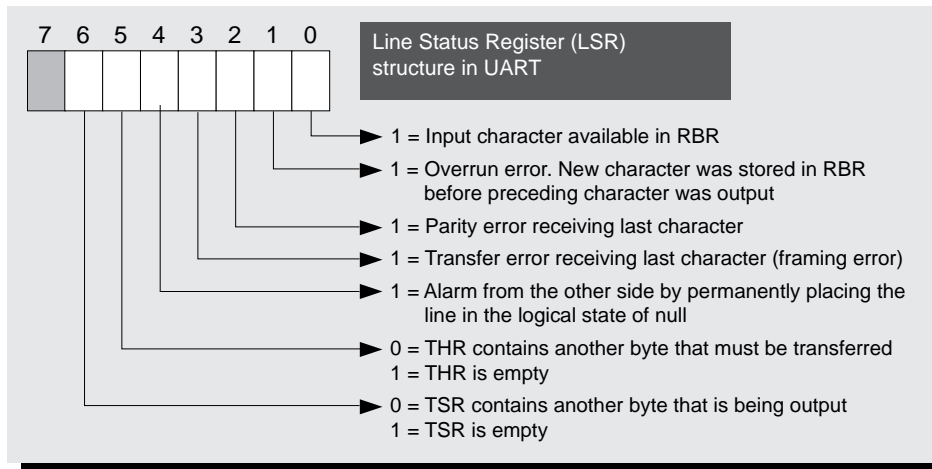
Current UART status

The line status register (LSR) contains the line status as well as the most critical information regarding sending and receiving of characters.

Bit 0 indicates whether a received character is waiting in the receiver register (RBR). This bit is automatically reset as soon as you retrieve the character from the RBR. Sometimes if this does not happen fast enough, another character comes in before the old one has been read. In such cases the old character is simply overwritten and gets lost. This condition is flagged by bit 1 of the LSR.

Two additional errors, parity errors and "overrun" errors, are covered by bits 2 and 3 of the LSR. The latter occur when the protocol (either data-bits, stop-bits or parity bits) is not maintained while receiving a character, generally due to line interference.

All errors, incidentally, are recorded by the UART without any corrective action being taken. It is the responsibility of the communications software, protocol and mechanism to inform sender and receiver of transmission errors and prompt them to retransmit the data in question.



When the sender issues an alarm through bit 6 in its line control register, it's reflected in bit 4 of the receiver's line status register. The receiver can then immediately respond.

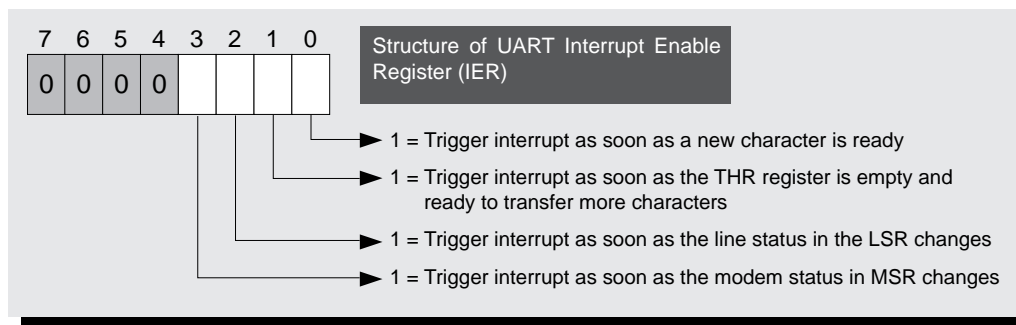
Bits 5 and 6 in the LSR give information about the current send status. They pertain to the two send registers, the transmitter holding register and the transmitter shift register. Bit 5 indicates whether the transmitter holding register is empty. If yes, the next character to be sent can be loaded into this register; if no, the software must wait until the bit goes to zero.

Also important is whether the transmitter shift register (TSR) is empty. If yes, it means the last character to be sent has been completely transmitted; if no, the character is still in the TSR and has not yet reached the receiver.

Control through interrupts

When interrupts are used to feed characters to the UART or to read incoming characters, the two interrupt registers are crucial. The interrupt enable register (IER) determines in which situations an interrupt should occur. The interrupt identification register (IIR) tells the serial port interrupt handler what event prompted the interrupt.

You must make the interrupt handler available yourself through software. The sample program SERIRQ which we've included on the companion CD-ROM and talk about at the end of this chapter shows how to build such an interrupt handler. To ensure proper calling of the handler, its address must be recorded in the interrupt vector for that particular serial port. For the first serial port this is IRQ4 with interrupt vector 0Ch, for the second IRQ3 with interrupt vector 0Bh. Not so well-defined are the third and fourth serial ports, where various settings are possible for the port address as well as the interrupt used. The software referring to serial ports above COM2 should always be designed so the user can tell it the port address and the interrupt. Unfortunately there is no fixed mechanism for detecting a serial interface with its port and interrupt.



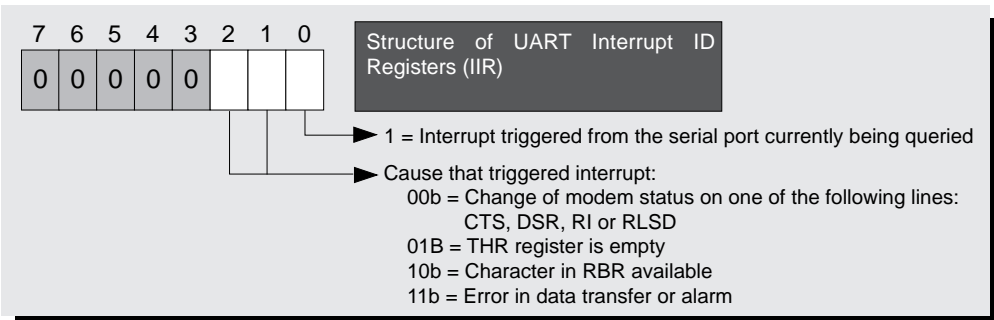
You selectively enter the events through the IER that will initiate a call to the interrupt handler. Set bit 0, for example, if the receiving of characters should follow through the interrupt handler and the handler is to be called whenever a new character is in the RBR. If the sending of characters will also occur within the interrupt handler, set bit 1 as well. In this case an interrupt will also occur whenever the transmitter holding register is empty.

It makes no difference whether you have set one or several interrupt sources using the IER. When calling the serial interrupt handler you should always check the interrupt ID register (IIR) first. Most important, you should inspect bit 0 of this register. Bit 0 indicates whether the serial port has actually executed an interrupt (which of course you can assume because the interrupt handler was called). What happens, however, when several serial ports use the same interrupt? First, you must determine which port has requested the interrupt. To do this, sequentially check the interrupt ID registers of the known ports and stop when you get to the port with a 1 in bit 0 of the IIR. You now have the port that has actually executed the interrupt.

Bits 1 and 2 in the IIR tell you why the UART executed the interrupt, so you can respond accordingly. For example, if code 01b is given, this tells the interrupt handler the next character can now be output to the THR. Note that we said "can" and not "must". UART simply sets the line to marking condition if it does not immediately need more characters. This indicates to the other side that at present it has no more characters to transmit.

On the other hand if bits 1 and 2 contain the code 10b when the interrupt handler is called, the handler should immediately read the receiver buffer register to obtain the received character.

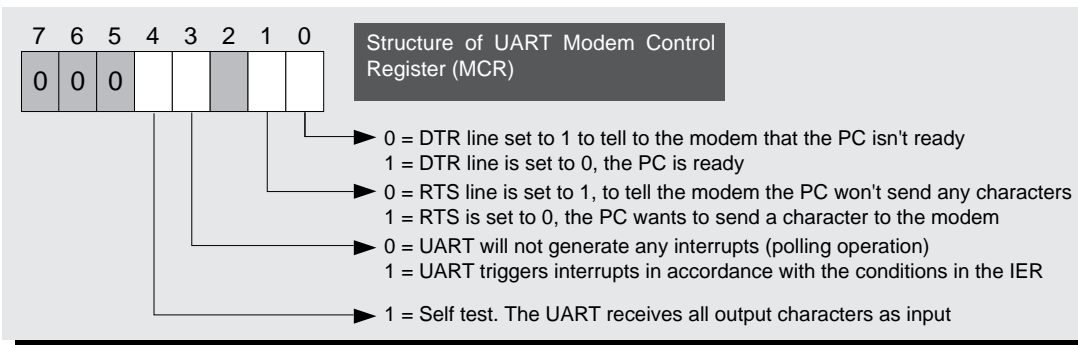
With an error or alarm (code 11b) the line status register should be read to determine the cause of error, which can then be routed through internal variables to the main program outside the interrupt handler. Code 00b is similar, except in this case the cause of the interrupt is determined through the modem status register.



Before the UART starts generating interrupts in the situations desired, you must not only set the IER, but also set a bit in the modem control register.

Modem registers

Just as the line status register and line control register oversee the connection between two serial ports, the modem status register and modem control register are responsible for the connection between a serial port and its attached modem.



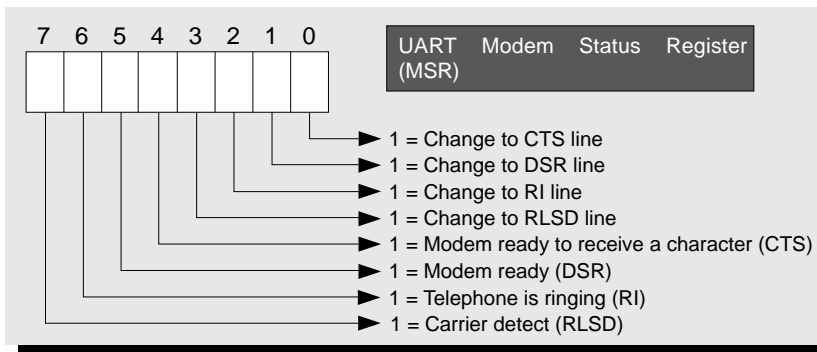
We've mentioned the modem control register contains a flag (bit 3), which can enable or disable interrupts through the UART. Also, communication between serial port and modem occurs through four control lines. Included in these four lines are the DTR (Data Terminal Ready) and RTS (Ready To Send) lines, which lead from PC to modem. These can be controlled through bits 0 and 1 of the modem control register.

According to the RS-232 protocol, the PC must first set the DTR flag to 1 so the DTR line goes to logical 0 and the modem knows the PC is ready. The PC must then set the RTS flag to 1 so the RTS line goes to logical 0 and the modem is made aware of the desired character transfer. In both cases the modem should answer by setting the appropriate handshaking signals.

Apart from the DTR and RTS, the UART can be switched to "loopback mode" through bit 4 of the modem control register. This mode helps in the development of programs that communicate through the UART with other devices. When this flag is on, the UART reroutes all output using THR directly to the RBR. An attached device is no longer required to test your program and respond to characters received. All you need to do is output characters to simulate receiving them. Even the interrupts will occur if interrupt mode is active.

When the DTR and RTS lines are set the modem generally responds through the DSR and CTS lines. You can check this using bits 4 and 5 in the modem control register, which always reflect the status of these two lines.

The modem status register can also be used to check the status of the RLSD and RI lines. Here bits 6 and 7 indicate whether a call is being received by the modem (RI) or whether a modem is present at the other end of the line (Carrier Detect).



In addition to bits 4 to 7, bits 0 to 3 also refer to the CTS, DSR, RI and RLSD lines. Rather than the current line status however, these bits indicate a potential change since the last read on the modem status register. By checking this register you can tell immediately if action is necessary - normally the case only when the status of one of the four lines has changed since the last inquiry.

Successors to the 8250

Microprocessor and controller manufacturers often suffer the same fate as software companies. They may be so anxious to release new products, the new chip may be released before it has been thoroughly tested. Bugs soon appear which result in grief, expensive recalls and public relations nightmares. This is what happened to National Semiconductor with its INS8250 chip, the original UART for PC's.

Shortly thereafter came a debugged version of the INS8250, the INS8250A, which was fully pin-compatible and function-compatible with its predecessor. Unfortunately however some developers of communications software had already based their products on the errors of the 8250, and their software no longer functioned with the A-version. National Semiconductor, therefore, felt obligated to develop another version of the chip. This new chip, called INS8250-B, intentionally included some of the bugs from the original version to guarantee compatibility with software already in use.

The NS16450 followed the INS8250-B. It was released with the AT. Although it was designed for the AT's expanded bus and interrupt capabilities, it remained fully compatible with previous versions. Today, however, most PCs have an NS16550A or NS16C552 instead of the 16450. The distinct advantage the NS16550A and NS16C552 have over their predecessors are two buffers for sending and receiving characters.

Send and receive buffers

The lack of such buffers severely limited the chip at high baud rates, especially in processing interrupts. Rates above 9600 baud were rarely achieved because the chip often received characters faster than it could route them to the software using interrupt. Computers were generally not yet fast enough and interrupt requests from the UART could not always be granted immediately. Finally, the serial interface being assigned to IRQ3 or IRQ4 does not have a high interrupt priority in the system as a whole.

Consequently, characters just received were frequently overwritten in the receiver data register by the next character. This resulted in an overrun error. In such cases the baud rate had to be reset to a lower value.

This was a much less serious problem with the NS16550A and its successor, the NS16C552. These chips have two 16 byte buffers, for the receiver buffer register (receive) and the transmitter holding register (send). These two buffers, which operate on the FIFO principle (first in, first out), have also given the 16550 its nickname of the "FIFO chip".

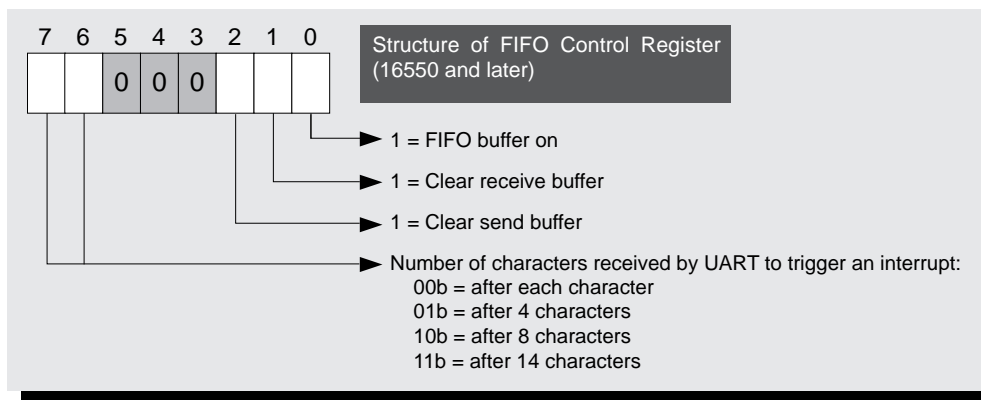
The send buffer is of lesser importance of the two buffers. When sending, the computer itself sets the pace and will not overburden itself. The receive buffer on the other hand is very important. You also need to involve the software of course to use this buffer. First, initialize the UART so the buffer will actually be used. Next, configure the character-reading routine.

When polling the port, little will change because normally here you would be checking the line status register to see if a character is available in the RBR. If so, you would read the character and begin a new pass through the loop. Then if any unread characters exist in the internal buffer, the UART will reset the corresponding bit in the line status register immediately after reading the character from the RBR. This way the character will automatically be captured in the next pass through the loop.

This is the type of loop missing from most serial interrupt handlers. As long as the UART had no characters in an internal buffer, it was enough to read the RBR once after an interrupt call and then wait for the next interrupt to retrieve the next character. With the buffer, a loop must be implemented within the interrupt handler, which continually reads characters from the RBR until the line status register indicates that no more characters are there. At this point all characters have been read from the internal buffer, because otherwise after reading the previous character the UART would have immediately moved any remaining characters from the buffer to the RBR.

To remain compatible with its predecessors, the internal buffers of the 16550 are disabled unless specifically instructed otherwise. A new register located at port address 2 relative to the base port of the interface is responsible for this task. Taking another look at the table on page 249 you'll see the interrupt ID register already exists at this address. Whereas the interrupt ID register is read-only, the new FIFO register can only be written to. This is why they can both share the same register address. The UART knows which register is meant based on the type of access.

Bit 0 of this register activates the FIFO buffers. In the meantime if you need to clear one of the two buffers, enter a value of 1 in bit 1 (for the receive buffer) or bit 2 (for the send buffer). Remember also to set bit 0 or the FIFO buffer will be activated at the same time.

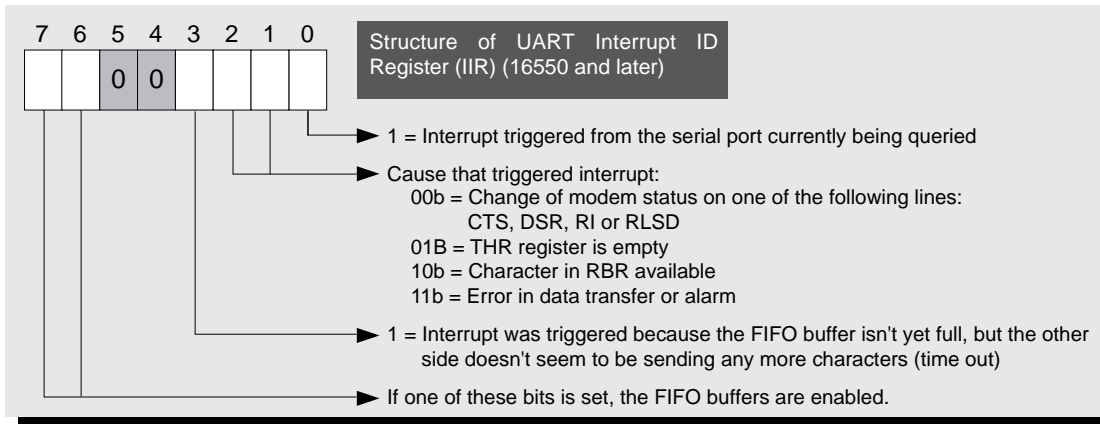


Bits 6 and 7 set the number of received characters after which an interrupt occurs if enabled through the interrupt enable register. The values 4, 8 or 14 in these bits stop the UART from generating an interrupt each time a character is received. This has a positive effect on software performance because interrupts take a relatively long time. The inquiry loop within the interrupt routine (discussed above) then sees to it that all characters are read from the buffer, not just the first.

Unfortunately, however, there is a "catch" to this method. Suppose the other side sends three characters and then stops the transmission because these three characters are telling the receiver to send back certain information. The receiver might not receive these characters because the buffer does not yet contain the required four, eight or fourteen characters after which only then an interrupt occurs. The sender could conceivably wait forever for the receiver's reply.

To prevent this from happening the UART always executes an interrupt even when the buffer is not yet full, but no new characters have arrived within the transfer-time for three characters. This "timeout" is communicated to the interrupt handler using an additional new bit in the interrupt ID register. This bit can normally be ignored however, because the UART also gives the reason for the interrupt - the availability of characters in the RBR.

When developing a communication program for the serial port, you should always include the FIFO options if they are available. The sample programs at the end of this chapter include a routine for identifying the type of UART installed.



Accessing The Serial Port From The BIOS

Since the serial port has become an essential component of the PC, the BIOS must deal with it as well. Unfortunately, built-in support for the programmer is weak. A total of four functions are available:

- Setting transmission parameters
- Checking line status
- Sending/receiving characters in polling mode

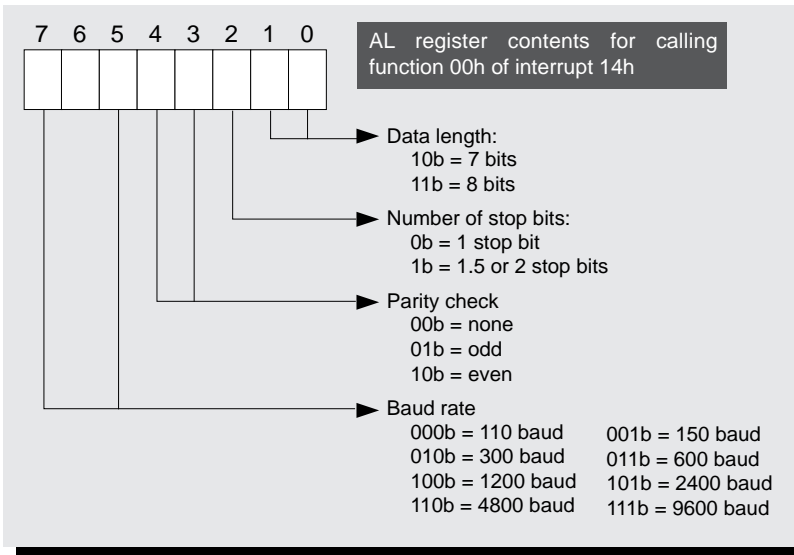
However, both support for interrupt-driven control of the serial port and functions to access the expanded capabilities of the 16550 and its successors are missing. In practice then, you're forced to program the various serial interface registers directly as we discussed in the previous section. The BIOS functions are included here only for completeness.

All BIOS functions for serial port access are called using interrupt 14h, with the number of the desired function passed in the AH register. In addition the DX register receives the port number, where 0 represents COM1, 1 for COM2, etc.

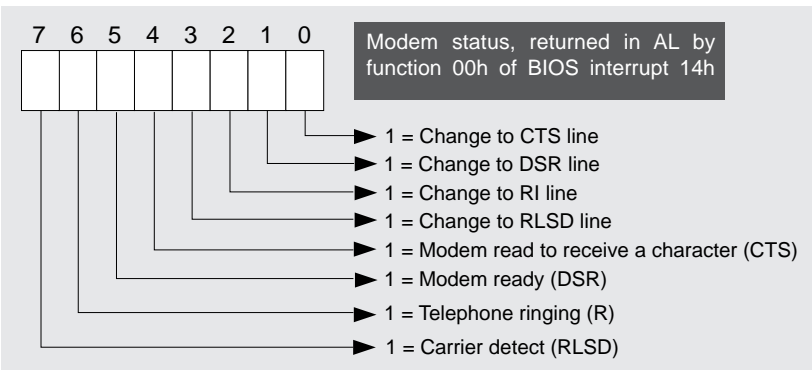
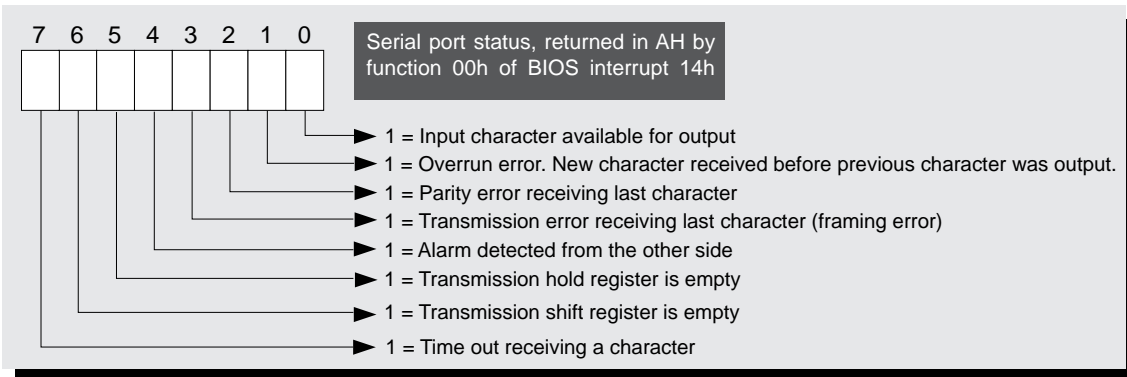
BIOS Functions For Serial Port Access	
Number	Task
00h	Set communication parameters
01h	Output characters
02h	Read in characters
03h	Query port status

Setting communication parameters and status check

In addition to the arguments in AH and DX, you must pass a value to function 00h in the AL register, whose structure is shown in the following illustration. Notice the BIOS permits only some of the settings directly supported by the UART. Word length for example can only be 7 or 8 bits, and the range of available baud rates lags far behind the capability of the UART.



Function 00h returns the line status from the UART line status register as its return value in the AH register. It returns the modem status from the modem status register as its return value in the AL register.



The line status (returned here in the AH register) can also be checked at any time by function 03h. Other than the function number in AH and the port number in DX this function requires no further parameters. As indicated above, it returns the port status in the AH register.

Sending and receiving characters

To send characters, use function 01h. In addition to function and port number, you must also pass the character to be transmitted in the AL register. If successful, bits 7 of the AH register is set to zero. A value of one means the character could not be transmitted. The remaining bits correspond to the line status.

Function 02h receives characters. After the function call the character received will be in the AL register. AH contains a value of zero if no error has occurred, otherwise its value corresponds to the line status.

You'll find the following program(s) on the companion CD-ROM



SERUTIL.H (C listing)
SERUTIL.C (C listing)
SERUTIL.PAS (Pascal listing)
SERIRQ.C (C listing)
ARGS.C (C listing / see Chapter 37)
WIN.C (C listing / see Chapter 38)
IRQUTIL.C (C listing / see Chapter 38)
SERIRQ.PAS (Pascal listing)
SERTRANS.H (C listing)
SERTRANS.C (C listing)
SERUTIL.PAS (Pascal listing)
ARGS.PAS (Pascal listing / see Chapter 36)
SERTRANS.PAS (Pascal listing)



The Parallel Port

Three ways are available to access the parallel port: Direct hardware programming, through the ROM-BIOS or with DOS function calls. In this chapter we'll discuss direct programming and using ROM-BIOS functions to access the parallel port. The first section describes the BIOS functions used in printing. The ROM-BIOS functions offer an advantage over equivalent DOS functions because they allow better control over printer status. DOS immediately fails when a printer triggers the critical error interrupt but BIOS offers other options.

In the second section of this chapter ("Direct Programming And The Parallel Port") we'll talk about direct programming of the parallel port. We'll show you how to connect two computers through their parallel ports and transfer data quickly between these computers using a file transfer program.

Accessing The Parallel Port From BIOS

BIOS interrupt 17H is reserved exclusively for communication with the parallel port. Most users call interrupt 17H the BIOS printer interrupt although other peripheral devices could also be connected to this port.

The BIOS printer interrupt

A maximum of three different parallel ports can be connected to the PC (see the "Direct Programming And The Parallel Port" section for more information). Interrupt 17H provides three different functions for addressing these ports. These functions perform three specialized tasks and can control all three parallel ports.

Function	Task
00H	Display characters
01H	Initialize printer
02H	Request printer status

About these functions

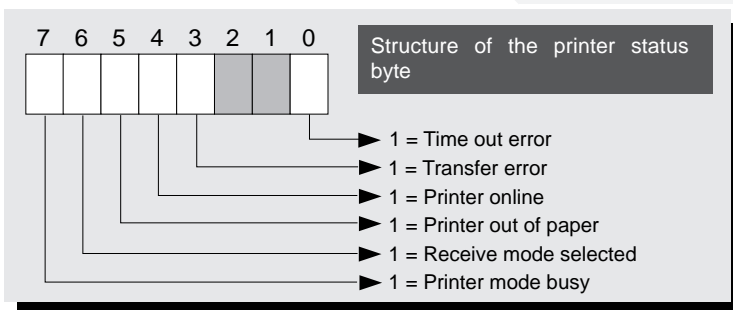
These functions are superior to the equivalent DOS functions in the choice of the addressed port. The DOS functions only control the first parallel port (PRN or LPT1). The three BIOS functions are more flexible in this respect and, when you call them, expect to find the number of the parallel port to be addressed in the DX register. You can specify values of 0, 1, and 2 for the ports: 0 corresponds to LPT1, 1 corresponds to LPT2, and 2 corresponds to LPT3.

Printer status

These functions have something else in common besides being passed the port number. After being called, each function returns the current printer status in the AH register. The bits of this status byte convey information about whether the printer is currently busy, still has paper, or has encountered an error while receiving characters. This status is very important to the communication with the parallel port.

Time out error

A time out error, signaled by bit 0 of the status byte, always occurs when the BIOS attempts to send data to the printer over a



certain period of time and the printer is BUSY (bit 7=0) or not accepting the data. This often happens because a parallel port can send up to 100,000 characters per second but no printers on the market can keep up with that pace.

The number of failures that occur before a time out occurs varies with the contents of a BIOS variable. Each parallel port has a byte allocated at a memory range beginning at address 0040:0078H. These bytes specify the number of unsuccessful attempts allowed.

BIOS time out counter for parallel ports	
Address	Meaning
0040:0078	Time out counter: first parallel port
0040:0079	Time out counter: second parallel port
0040:007A	Time out counter: third parallel port

Instead of referring to a given period of time, the values contained in these variables refer to the number of failed attempts that are allowed before BIOS reports a time out error. The program code of the ROM-BIOS continually prompts the parallel port within a program loop. Since this loop consists of only a few assembly language instructions running in cycles of a few microseconds, the time out value from each BIOS variable acts as a factor used in the loop's counter, instead of as the loop counter itself. This factor is multiplied by the constant 262,140 (4 x 65535). The value 20, which the BIOS enters in all three memory locations after the system boots, corresponds to more than five million attempts.

If you use a loop counter instead of a time unit, the period of time that can elapse before a time out occurs depends on the processing speed of the computer. This means that the time span varies with the system's CPU and clock speed. That's why the loop counter must be increased on faster systems, because there isn't a connection between the printer speed and the CPU speed. If you don't make this adjustment, you'll discover, after purchasing a 486, that the system will suddenly send a time out error message during printing.

The BIOS manufacturers usually make this adjustment by using a larger constant instead of a larger default value. For example, if your new computer is twice as fast as a normal AT, instead of 4x65535, you might multiply 8 by 65535. This is done because applications also access the three BIOS variables to change the time out rate for one of the ports. This is possible because these variables are accessible to any program as part of RAM. This gives the programmer the option of setting a higher time out rate for situations in which the printer would otherwise send a time out error. However, increasing the time out rate wouldn't work on a faster system, unless a large enough constant factor is also chosen.

Other printer status flags

Other bits provide more information about the printer's status. Bit 3 shows a transfer error (a data error in the line), while bit 4 indicates whether the printer is currently online or offline. This bit is the equivalent of the online button found on printers, with an LED indicating its status.

Bit 5 indicates whether the printer has any paper. Bit 6 confirms the printer's receipt of the last character. To determine whether a printer is connected to a particular port, simply prompt for this bit. If it contains a value of 1, then a printer exists.

Bit 7 represents the BUSY signal. It's used by the printer to indicate that it's busy and cannot accept any characters. This bit is also important to the time out error. This signal instructs the ROM-BIOS to repeat the output loop because a character cannot be sent to the printer. This is negative logic: If this bit is set to 0, the printer is busy, and if it is set to 1, the printer is not busy.

Different states of the printer can also result in changes to a series of bits in the status byte. For example, if the printer is ready to print and is online, bits 7 and 4 are set. If you switch the printer offline, not only are bits 7 and 4 cleared, but bit 3 is also set, which signals a transfer error.

Checking printer status

You may be wondering how you can use this status byte in programs. First, the status byte can prompt for the various states that correspond to the single bits before or after transferring a character. This means that you can determine whether the printer is out of paper, switched offline, or connected to the parallel port.

The status byte can also be used to check for printer access. If bit 1 (time out error), 3 (transfer error), or 5 (out of paper) is set, or bit 4 (printer online) or 7 (printer busy) is cleared, you cannot send characters to the printer. The following pseudocode demonstrates how this is done:

```

pstatus = PrinterStatus;
if ( ( (pstatus and 29h) <> 0 ) or
      ( (pstatus and 80h) = 0 ) or
      ( (pstatus and 10h) = 0 ) ) then
  PrinterOK = FALSE
else
  PrinterOK = TRUE;

```

Now let's return to the three BIOS printer interrupt functions.

Function 00H:	Write character
---------------	-----------------

Function 00H writes a character to the printer. Place the function number (00H) in the AH register and the ASCII code of the desired character in the AL register. After the function call, the AH register receives the status byte.

Function 01H:	Initialize parallel port
---------------	--------------------------

Function 01H initializes the parallel port and printer. Always execute this function before sending data to the printer. Place the function number (01H) in the AH register. After the function call, the AH register receives the status byte.

Function 02H:	Get status byte
---------------	-----------------

Function 02H reads the status byte. With this function, a printing job and the initialization of the parallel port aren't involved. After the function call, the AH register receives the status byte.

Calling the BIOS functions

Each of these functions can be called from a high level language program in the same way you would call any interrupt. Some C compilers support these functions through their runtime libraries. The table on the right shows the corresponding routines with Borland and Microsoft compilers.

Although QuickBASIC and Turbo Pascal also have printer support, this support uses DOS output functions instead of BIOS functions. If an output error occurs, the DOS functions call critical error interrupt 24H instead of returning an error code to the program. However, this error can be intercepted (e.g., by Turbo Pascal).

QuickBASIC contains the LPRINT statement for transmitting printed output. Turbo Pascal uses Write and WriteLn for the same purpose, provided that the programmer opens a file variable, directs this variable toward the printer, and specifies the printer before calling WriteLn or Write. The PRINTER.TPU unit included with Turbo Pascal performs this task for you (refer to your Turbo Pascal documentation for more information).

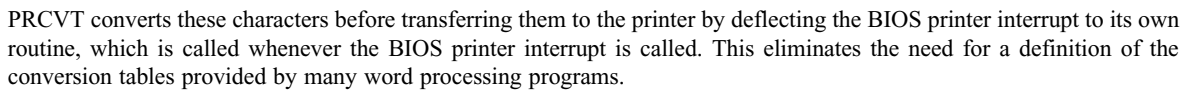
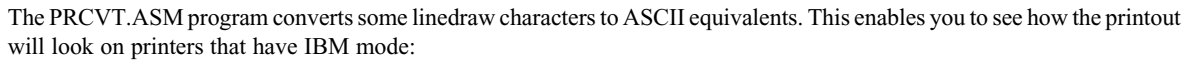
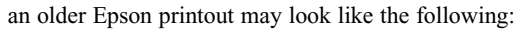
C Compiler Support for Printer Functions	
Compiler	Function
Turbo C	
Borland C	
Borland C++	biosprint
Microsoft C	
QuickC	bios_printer

Redirecting the BIOS printer interrupt

TSR programs redirect the BIOS printer interrupt to their own routines to suit the needs of these TSRs. This allowed the development of print spoolers (programs that intercept characters sent by the original BIOS functions and store these characters in a buffer for later printing).

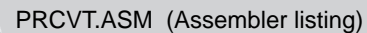
Demonstration program

The PRCVT.ASM assembly language program on the companion CD-ROM will help users whose printers uses a different character set than their PCs. For example, if you attempt to print a program listing or file containing PC linedraw characters on some older model Epson printers, the printout may look different than you expected. If the data on the screen looks as follows:



The new interrupt handler, which is the focus of the PRCVT.ASM TSR program, first checks whether function 00H should be called. This is the only function to be changed. If another function is called, the call is passed to the old printer interrupt. If a character should be output, the program checks a table called CODETAB to determine whether it contains this character. This table, which you can see at the beginning of the program listing of PRCVT.ASM, consists of 2-byte entries, with the first byte (the low byte) containing the new code of a character that will be converted, while the subsequent byte reflects the character's old code. The table is closed by a byte with the value 0.

You'll find the following program(s) on the companion CD-ROM



This program can be used for both BIOS and DOS printed output.

If the receiver can keep up with the sender, the BIOS functions for parallel port character output work efficiently. Communicating with a printer is the safest method, but linking two computers through their parallel ports is more complicated. This often requires data transfer rates that extend beyond the capabilities of the BIOS functions. A special type of cable with different pin assignments (called a parallel transfer cable) is needed to connect two computers. The BIOS functions cannot be used with this type of cable because they assume that the normal assignments are being used for each line in the cable.

The I/O ports

Up to three parallel ports can easily be installed in your computer. The I/O address space reserves three ranges for parallel interfaces (see the table on the right). The port addresses in the previous table are listed in the sequence in which BIOS examines them on startup, instead of in numerical order. From this table, BIOS determines which port addresses are LPT1, LPT2, and LPT3.

Port	Interface
3BCH - 3BFH	Parallel interface on MDA card
378H - 37FH	Parallel interface 1
278H - 27FH	Parallel interface 2

The BIOS begins by checking the block at address 3BCH-3BFH. This is part of a large address block that extends from 3B0H to 3BFH, and is reserved for a Monochrome Display Adapter (MDA) or Hercules Graphics Card. During the 1980s most PCs were delivered with this type of video card. In addition to the video logic, these cards included a parallel port.

If the BIOS finds a video card with a parallel port, the BIOS addresses that parallel port as LPT1. The next parallel port found will then be LPT2. If the video card doesn't have a parallel port, then the first parallel port located will be identified as LPT1.

The other two address blocks listed in the table are for additional parallel ports. These ports may exist on two different cards, or on a single I/O card. Regardless of how the hardware for the parallel port is detected, the BIOS checks for the existence of parallel ports according to the previous table. Suppose that only one parallel port is installed, but it's using the address block reserved for the second. This port will still be recognized by the BIOS as LPT1.

Assigning LPT1, LPT2, and LPT3

The BIOS assigns the names LPT1, LPT2 and LPT3 to the parallel ports by entering their base addresses as variables in the BIOS variable segment. A four-word array starting at offset address 0008H of this segment retains the port addresses of the parallel ports (see the table on the right).

Address	Contents
0040:0008H	Base address of LPT1
0040:000AH	Base address of LPT2
0040:000CH	Base address of LPT3
0040:000EH	Base address of LPT4

The variable segment can accommodate four parallel ports, even though the BIOS will only look for three parallel ports when you start your system. The BIOS functions will also let you work with a fourth parallel port if you enter its base address by hand at offset address 000EH in the BIOS variables segment. The BIOS then addresses it as interface 3.

The LPT terminology originates from DOS rather than BIOS. LPT1 represents interface number 0, LPT2 represents interface number 1, etc. (DOS doesn't recognize LPT4).

If you want to change interface numbers (e.g., have DOS send output intended for LPT2 to LPT1), you must change the port addresses in these three BIOS variables. The pseudocode for this example would look similar to the following:

```
DummyWord = MEM[ 0040H: 0008H ]
MEM[ 0040H: 0008H ] = MEM[ 0040H: 000AH ]
MEM[ 0040H: 000AH ] = DummyWord
```

Port registers

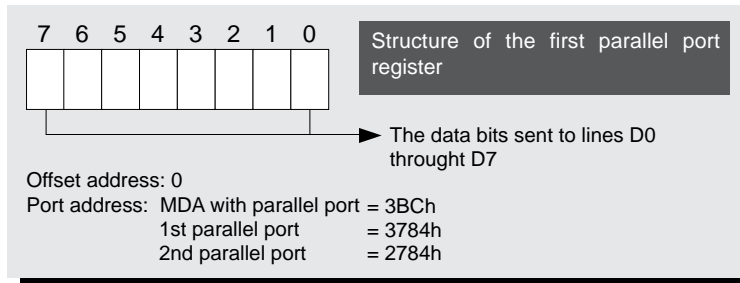
Regardless of their locations in the addressable memory, all parallel ports use the same register interface, which consists of three ports. These ports occupy the first three port addresses of the card (e.g., 378H, 379H and 37AH) for the first parallel port. The following illustrations show the meanings of each bit in the port registers. When you compare the assignments of each bit in the tables to the structure of a parallel cable, you'll see that they mainly coincide. A direct connection exists between the bits of the port registers and the lines in a parallel cable. When a bit in one of the registers is set to 1, then an electrical signal is immediately sent over the corresponding line. If the bit value is set to 0, then the current in the line returns to "low" status. The current in the line will always reflect the status of the corresponding register bit as it is manipulated by the software.

Some of the lines in the cable use negative logic. These lines have names preceded by minus signs. The condition associated with this type of line will be executed if the corresponding bit has a value of 0. For example, the ERROR line indicates a problem with printer output only if the corresponding bit is 0. As long as this bit remains set to 1, no error will be indicated.

Data lines

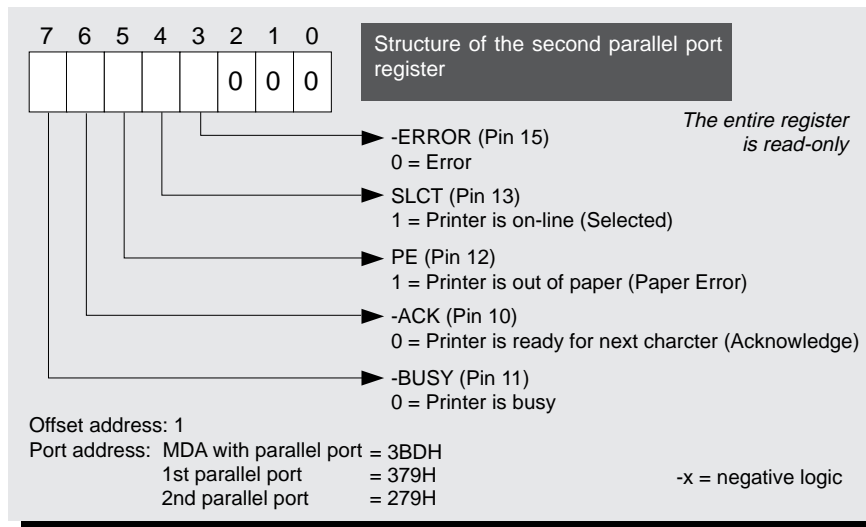
The eight bits of the first parallel port register use positive logic. This register stores the eight data bits that will be carried along data lines D0 to D7 and transferred to the receiver. Remember that this register was intended to be only an output register on a parallel port. It wasn't designed to receive data.

Remember that printers don't send data back to their hosts, and this type of port was never intended to be used for connecting two computers. This will cause some problems when you're developing a communications program because you must deal with communication between two computers as sender and receiver (more on this later).



Printer status

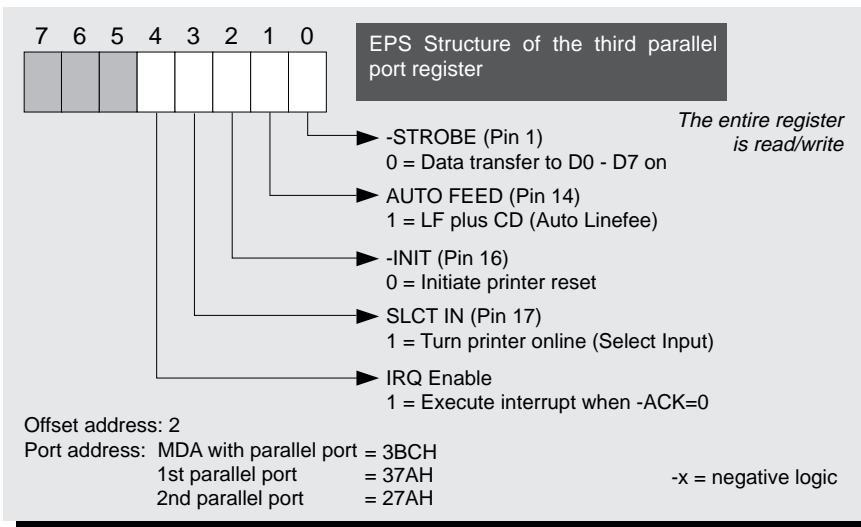
The second register is responsible for the current printer status and is read-only. This register reflects the condition of the status lines coming from the printer. The following illustration names the pins from the host's point of view.



Printer control

The third register controls the printer and its hardware, and plays an important role in transferring characters. Except for bit 4, all bits are connected to corresponding pins of the parallel port.

A bit hidden in this register can execute a hardware interrupt as soon as the ACK signal switches to low, which indicates that the printer has received the last character. You can usually determine which interrupt will be executed by setting some DIP switches on the port. You can choose between IRQ 7 and 5, which are associated with interrupts 0FH and 0DH. Unlike serial ports, this option is rarely used with parallel ports because these ports work on the polling principle instead of the interrupt principle. This also applies to the BIOS, which doesn't use this interrupt vector.



Communication between printer and host

The assignment of each pin in the port and the meanings of the corresponding register bits become apparent when you take a "behind the scenes" look at communication between host and printer. First, the byte sent out by the host passes to the first parallel port register and is transmitted through data lines D0 through D7. This signal immediately arrives at the printer but more information is needed before this first byte can be processed. Since there is always some sort of signal coming in on data lines D0 through D7, the printer doesn't know whether this is the first character to print or simply a stray byte from the last transmission.

The -STROBE line

The -STROBE line is important for keeping track of data. When the host sets this bit to 0 (and thus setting the current in the corresponding line to low), the printer knows that a character is coming over the data lines. The host must then disable the -STROBE signal quickly; otherwise the printer may read the character twice. The printer hardware needs only a microsecond to read the character from the data lines.

The BUSY line

Since a microsecond isn't a very long time, the printer would never be able to keep up with this kind of data transfer rate, even if it stored the characters in an internal buffer. The BUSY line pauses the communication long enough to process the character it has just received. A BUSY signal is generally sent immediately after a -STROBE signal.

The software or ROM BIOS must then wait until the printer removes the BUSY signal before it can send the next character. The BUSY line is the only pin in the parallel port that inverts the signal when it's received. In order for the host to receive 0, the printer must send a value of 1 over the BUSY line.

The -ACKnowledge line

The printer must also send an -ACK signal of 0 on the -ACKnowledge line. Because of the negative logic of this line, the host will receive this as a value of 1, which indicates that the printer received the character that was sent.

The durations of all signals needed to transmit one character add up to about 10 microseconds. Theoretically, this would produce a data transfer rate of 100,000 characters per second. However, in reality, processor overhead adds a lot of extra time. Real transfer rates are actually about 1/100th of this (1000 characters per second), even if the printer has its own buffer for storing characters as they are received.

The printer responds

Although communication between a host and a printer is mostly unilateral, the printer does offer feedback to the host. The printer uses three pins to send information back to the host: -ERROR, SLCT, and PE. All three of these pins have their corresponding bits in the first parallel port register.

SLCT represents "Select". This corresponds to the ONLINE switch found on the front of your printer. If you turn the printer offline, the printer will signal the host using the SLCT line.

PE represents "Paper Error". This allows the printer to tell the host that it's out of paper or that the paper feed is jammed. This type of error is separated from normal data transfer errors, which are transmitted through the ERROR line. This is done because paper errors can be immediately corrected by the user but data transfer errors are more serious. Data transfer errors are usually caused by cable failures or electrical disturbances.

Host control

Obviously, the host has some control signals that it uses to command the printer. These signals are -AUTO FEED, -INIT, and -SLCT IN. The bits that receive these signals are found in the third parallel port register, where the values can be read or manipulated by software.

-AUTO FEED tells the printer to add a linefeed to every CR (carriage return) character (ASCII code 13) it receives as long as this signal is set to high (1). This line is included because all printers don't react the same way when they receive a CR (carriage return) character. Many printers simply return to the start of the current line without adding the linefeed, which moves the print head down to the next line. So, the LF (linefeed) character must be added separately.

The host can use the -SLCT IN line to turn the printer OFFLINE by sending a signal of 1. Normally, this line will be set to low so the printer stays online.

The host can use the -INIT line to reset the printer. To execute a reset, set the corresponding bit briefly to 0, and then immediately back to 1. If you don't set the value back to 1, the printer will reset itself repeatedly.

Using the proper cable

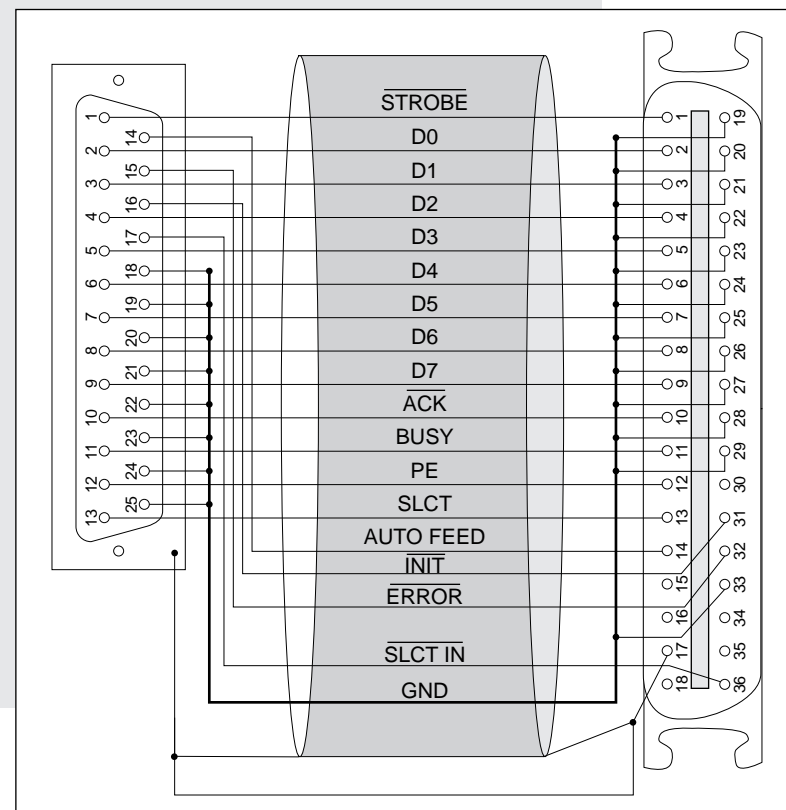
The entire transfer of data between host and printer will only work if the correct pins on the two ports are connected by a proper cable. Which signals are found at which pins and the way in which the pins are connected is standardized. The Centronics standard describes both the pin assignments at each port and the lines in the cable.

The illustration at the top of the following page shows how the pins of the host and printer ports are connected. The second illustration shows the structure of a parallel cable with the ground line.

*Cable connections
between parallel
port and printer*

Computer pin	Printer pin	Signal name	Meaning
1	1	-STROBE	Indicates data transfer
2	2	D0	Data line - bit 0
3	2	D1	Data line - bit 1
4	2	D2	Data line - bit 2
5	2	D3	Data line - bit 3
6	2	D4	Data line - bit 4
7	2	D5	Data line - bit 5
8	2	D6	Data line - bit 6
9	2	D7	Data line - bit 7
10	10	-ACK	Last character received
11	11	-BUSY	Printer busy
12	12	PE	Printer has no paper
13	13	SLCT	Printer is online
14	14	-AUTO FEED	Automatic CR after LF
15	32	-ERROR	Data transfer error
16	31	-INIT	Reset printer
17	36	SLCT IN	Turn printer online
18-25	19-30	GND	Ground

*Structure of a
parallel cable*



A do-it-yourself parallel transfer cable

If you want to "misuse" your parallel port for transferring data between two computers, a normal parallel cable won't work. One problem is that the parallel ports on both computers have identical female connectors. So, one end of the parallel cable won't plug into a second computer.

Another problem is that normal parallel communications travel in only one direction. One computer can use data lines D0 to D7 to send data, but it cannot receive data over these same lines and the other computer cannot send data over them.

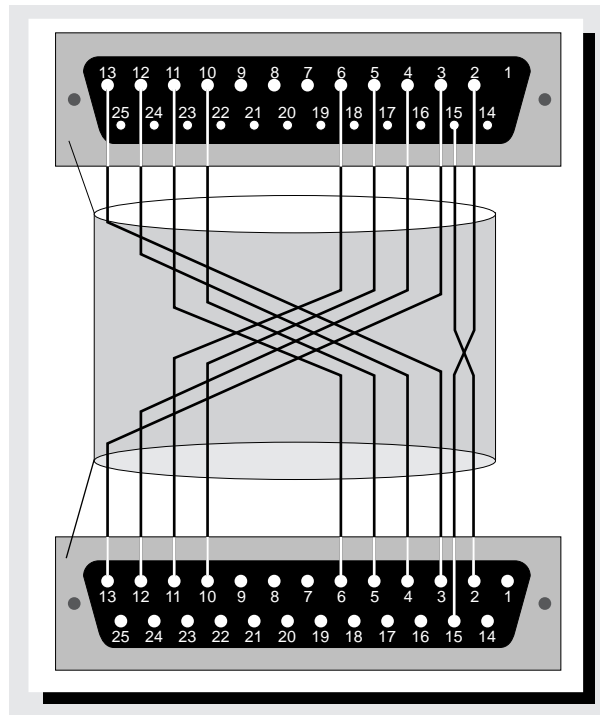
Usually data transfer between two computers requires a bidirectional connection. For example, the receiver will return a checksum of the data it received, so the sender will know whether the data was received without error.

The status lines used by a printer to return status information to the computer can provide a solution. These are the -ERROR, SCLT, PE, -ACK, and BUSY lines, which are associated with the second parallel port register. These lines are connected to data lines D0 to D4. This means that the receiver reads output from the sender through the status lines previously listed. Conversely, data lines D0 to D4 from the receiver are connected to the status lines of the sender, which enables two-way communication.

So, basically we're simply crossing data lines D0 - D4 with the -ERROR, SLCT, PE, -ACK, and -BUSY status lines. So, the following rule applies to both sender and receiver: Any data sent out via the first five bits of the first parallel port register will be received by bits 3 to 7 in the second register of the other communication partner. It doesn't matter which end of the cable is connected to the sender and which end is connected to the receiver.

The following illustration shows which pins to connect at each end of the cable to make a parallel transfer cable.

*Pins to connect
when you want
to make a
parallel transfer
cable*



This type of cable is very difficult to find commercially unless you own a LapLink or similar cable. If you want to make your own, you need the following information:

You'll need two male DB-25 connectors and a shielded single-pole cable less than 10 feet in length. Parallel cables longer than this cause data transfer problems.

As the illustration on the right shows, you must connect pins 2 to 6 on one connector with pins 15 and 13 to 10 (leave pin 14 free) on the other. Solder any five lines from the cable to pins 2 to 6 on the first connector. Then connect the other end of each wire to the proper pin on the other connector. Be sure to follow the proper order. For example, D0 must be connected with -ERROR, not SLCT or BUSY.

Now you must repeat the entire procedure for the other side of the cable to cross the connections properly. Don't forget to solder the cable shielding to the connectors as a ground.

*Pin connections
for a parallel
transfer cable*

Pin		Pin	
2	15	15	2
3	13	13	3
4	12	12	4
5	10	11	6
6	11	10	5

This type of cable can be used with commercial data transfer programs, such as LapLink. These programs usually work with the same type of cable as our parallel transfer cable. If the cable doesn't work, check your pin connections again. It's also possible that the program assumes the data and status lines are connected in a different order. A parallel transfer cable can be used to connect two PCs and transfer data between them. You can also use this type of cable to control a slave PC from a master PC.

Sample programs

The following file transfer programs listed are named PLINKP.PAS and PLINKC.C. Both can act as a sender or receiver in parallel file transfer. The basic syntax for calling either program is as follows:

```
PLINKP
PLINKC
```

The operating mode depends on how you start the program. The previous syntax sets the program in receive mode. The receiver waits until the sender begins transmitting or until the user presses **Esc** to exit. If a sender doesn't appear, the receiver waits until a time out error occurs, then exits.

The following examples start the program in sender mode, and try to send FILENAME.EXE to the receiving computer:

```
PLINKP FILENAME.EXE
PLINKC FILENAME.EXE
```

The sender waits until it senses a receiver. If a receiver exists, file transfer begins. If it doesn't, the sender waits until it recognizes a receiver or until the user presses **Esc** to exit. If a receiver still doesn't appear, the sender waits until a time out error occurs, then exits. You can also use wildcards to specify entire groups of files for transfer. The following examples start the program in sender mode, and try to send all EXE files to the receiving computer:

```
PLINKP *.EXE
PLINKC *.EXE
```

The two optional switches /P and /T can also be entered when you start the program. The /P switch specifies the parallel port through which you want information sent other than the default (LPT1). A number between 1 and 4 must follow the /P. The following examples start the program in receive mode and configure LPT3 as the parallel port for receiving data:

```
PLINKP /P3
PLINKC /P3
```

The /T switch specifies the number of time out intervals. A single time out is 10 seconds; you can enter any group of 10 second intervals. Enter a number after the /T switch. The program multiplies this by 10. The following examples start the program in sender mode, request a 30 second time out, and attempt to send all TXT files:

```
PLINKP /T3 *.TXT
PLINKC /T3 *.TXT
```

You can get help by typing PLINKC or PLINKP and the `/?` parameter. Enter the following to see the command syntax:

```
PLINKP /?
PLINKC /?
```

The program lists the switches for interface and time out intervals. Since these programs are coded, we omitted some features, such as checking for existing files. If you try to transfer a file to a receiver containing a file of the same name, the programs overwrite the existing filename, then time out. Check your receiver before sending files or add your own code to check for files.

Transferring data over the parallel transfer cable

With a parallel transfer cable like the one we described, you can simultaneously send five bits through data lines D0 to D4. You can transmit data in both directions simultaneously because the connections are crossed. However, some kind of communications protocol is needed so data can be transferred systematically. We need something similar to the -STROBE line to keep track of the data transfer pace. One of our five data lines must be used for this purpose.

The BUSY bit seems to be best suited for this job. The two outer bits (-ERROR and BUSY) are actually the only possibilities because the four data bits must be next to one another. Of these two, BUSY will be a better -STROBE line because it has no real application of its own for data transfer. Also, this bit is automatically inverted by the hardware. If you used this bit for data, it would have to be inverted after the transfer, which would be time-consuming. The inversion doesn't affect a -STROBE bit. Actually, it's important for the communications protocol. A 0 at one end of the cable must come out as a 1 at the other end. We'll discuss this in more detail later.

Communications protocol refers to the two demo programs described in this section. Although there can be numerous communications protocols, in this instance we're being quite specific. The protocol used here works on two levels, the byte level and the data block level. Each of these levels is handled separately. The data block level is on top of the byte level. The byte level is hardware-oriented, but the block level works with the software.

Data transfer at byte level

First let's discuss the byte level from the sender's point of view. The first thing we must consider is that an entire byte cannot be sent at once. A byte consists of eight bits and we can only send four bits in one direction at one time. So, each byte is divided into two halves called *nibbles*, which are sent one after the other.

First, the low nibble of the byte is written to bits 0 to 3 of the first parallel port register. From here it is sent out through data lines D0 to D3. The bit for data line D4 is set to 0 so the receiver will receive a value of 1 at its BUSY pin. This will tell the receiver that the low nibble of the next byte is ready to be read. This means that the receiver simply waits and reads the status line until the BUSY bit contains a value of 1.

The BUSY bit is then no longer useful, so the receiver proceeds by reading the nibble from the corresponding bits of the second port register. The contents of these four bits are stored in a variable and sent back to the sender through the data lines. The bit for data line D4 is set to 0, so a value of 1 is received back on the other end. This indicates that the returned nibble can be read and saved.

When both nibbles have been returned in this way, the sender can determine whether the byte was properly transferred. The communication is therefore verified at the byte level. However, this isn't the usual procedure. Since it takes too much time to check each byte individually, this type of verification is normally performed only at the block level. In our case, this argument doesn't apply, since the BUSY line must be used to send a -STROBE signal back to the sender with each nibble anyway. It doesn't take any more time to send the entire nibble back.

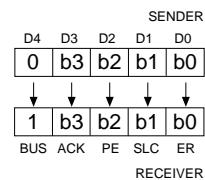
The transmission of the second nibble is basically the same, except that the value of the status bit is changed to 1. So, the receiver will receive a value of 0 at the BUSY pin. The receiver has been waiting for this as the signal to read the high nibble.

The nibble is then returned to the sender and the sender's BUSY bit is reset to 0. The two nibbles can then be combined to form the complete byte, and the transfer is complete from the receiver's point of view. The sender combines the two nibbles sent back by the receiver and checks the complete byte for data transfer errors. The program's send routine alerts its caller of any errors so the appropriate action can be taken and the data block can be resent if necessary. Most data transfer error can be detected in this way; exceptions include unusual types of cable interference.

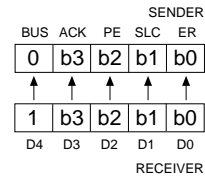
As we saw with normal communication to a printer, successful communication between two computers requires the proper switching of the -STROBE signal over the BUSY line. Remember that because of the way the lines in the cable are crossed, we're dealing with two separate BUSY lines. The same applies to both sender and receiver: For output, the BUSY bit is data line D4. The signal is received, however, at the BUSY status line on the other side.

The communications protocol at the hardware oriented byte level

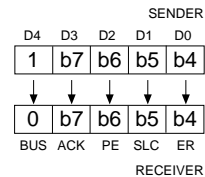
- 1 The sender begins the data transfer with the low nibble. It writes these bits to data lines D0 - D3 and sets the value of D4 to 0 so the receiver will get a value of 1 at its BUSY pin.



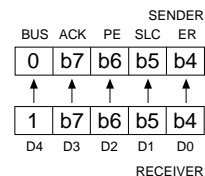
- 2 The receiver has been waiting for the value of the BUSY bit to change to 1. It then writes the nibble it received to data lines D0 - D3 to return it to the sender. To indicate it has received the nibble, the D4 bit is set to 0 so the sender's BUSY bit will change to 1.



- 3 The sender has been waiting for its BUSY bit to change to 1. It then writes the high nibble to the data lines and sets D4 to 1. This changes the value of the receiver's BUSY bit to 0.



- 4 This time, the receiver has waited for the BUSY bit to change its value to 0. The received nibble is again returned to the sender and the data bit D4 is set to 1. The sender will then get a value of 0 for its BUSY bit.



- 5 The communication is completed. The receiver reassembles the byte from the two nibbles and the sender checks the data returned from the receiver to make certain there were no data transfer errors.

Sender and receiver both use the data lines D0 - D4 for sending information and each has its own separate -STROBE line. This is shown in the previous illustration.

Time out problems

Communications protocols usually function without error as long as the electrical current isn't interrupted. If an error occurs, either the sender or receiver will be left waiting for the other end to respond to its last message. To prevent a situation in which

one of the communications partners is waiting forever for an answer that may never come, a time out value is set. The time out value determines how long one of the systems will wait for an answer from its partner before terminating the connection.

We mentioned in the first section the BIOS also uses a time out counter for communicating with the parallel port. The time out interval is usually measured by executing a read loop a certain number of times. For a program that must run on different PC systems, this isn't easy to manage. The time needed to process the read loop can vary with the system's processor speed.

Here is an example of how the time out counter works. Suppose that the sender has just sent the low nibble of a byte. It sets the time out counter to its maximum value and then waits for its BUSY bit to be set to 1 by the receiver. The read loop then begins to execute. It will continue to run until either the value of the BUSY bit changes to 1 or the time out variable reaches 0. The time out variable will continue to count down as long as its value isn't equal to 0.

This would look as follows in pseudocode:

```

TimeOutCount = MAXVALUE
WHILE ( BUSY-Bit = 0 ) AND ( TimeOutCount > 0 ) DO
    BEGIN
    END

IF TimeOutCount = 0 THEN
    error
ELSE
    o.k.
END

```

The communications protocol is activated along with the time out reading in both demo programs with routines called SendAByte and ReceiveAByte.

Synchronization

Once the communications protocol is activated, it will work without any problems. However, sometimes getting it started can be a problem. Before communications begin, both sender and receiver must have a value of 0 in the BUSY bit. If this is not the case, the receiver will immediately assume that a nibble has already been sent and it will try to read it, although the sender hasn't even sent anything yet.

The sender and receiver must be synchronized before communications can begin. Determining whether the sender or the receiver should be started first is complicated. For the demo programs presented here, we'll use the receiver as our starting point.

For initialization, the receiver waits for the sender to set its BUSY bit to 0. It then sets the sender's BUSY bit to 0. The synchronization is complete when the BUSY bits on both sides are set to 0. In the demo programs, this is done within the PortInit routine. A time out limit is also used in the initialization procedure. It works according to the principle previously described. In addition, the programs enable the user to quit at any time by pressing the **Esc** key. Otherwise, you may have to wait several minutes for the timeout interval to be reached.

Stopping the program

Both programs include a keyboard interrupt handler that is activated by pressing the **Esc** key. As with the timer interrupt handler, the program also uses a variable to communicate with the keyboard interrupt handler. In this case, it's a variable of type BOOL, which is set to TRUE when the **Esc** key is pressed. We can avoid having to read this variable separately in the read loop by coupling it with the time out variable. This is done by setting both the escape variable to TRUE and the time out variable to 0 when the **Esc** key is pressed.

The program will then simply respond as though the time out interval has been reached. A quick check of the escape variable will then allow you to determine the cause of the interrupt.

Block level protocol

The block level is above the byte level. As the name suggests, the block level is used for transferring entire data blocks from sender to receiver. This is strictly a software protocol. It's independent of the hardware because it relies on the send and receive routines from the byte level. In our demo programs, these are the SendABlock and ReceiveABlock routines.

A block always contains the following information: A token that precedes the block and describes its contents, the length of the block, and the block itself. The token is used so the receiver can immediately recognize what is being sent without having to read the data block. According to this convention, the SendABlock routine expects to be passed the token, the number of bytes in the block, and a pointer to the data block itself. Both the token and the number of bytes are handled as a sort of header and kept separate from the actual data block. The receiver must first correctly receive the header before the first byte of the data block will be sent. Imagine what would happen if the sender wanted to send a 120 byte data block but the receiver was expecting a block of 200 bytes. The receiver would count the next 80 bytes as part of the first data block, and the communication would be hopelessly tangled.

Remember, at the byte level, the receiver is sending every byte it receives back to the sender. So, the block level protocol will immediately know whether the header was properly transferred. Unfortunately, this doesn't let the receiver know whether it received the header correctly. The sender therefore notifies the receiver by sending a standard character. So, the receiver doesn't have to assume that there weren't errors in the header.

As feedback, the sender will send an ACK character (Acknowledge) character if the transfer was successful, or a NAK character (Non-Acknowledge) character if there was an error. The ACK character isn't related to the port pin of the same name; it simply fulfills the same function. The ACK and NAK characters are represented by the codes 00H and FFH in the demo programs, but you can use any codes. These characters also play a part in the communications protocol, since they are also checked at the byte level for successful transfer.

When the receiver has received the header and the subsequent ACK character without errors, the sender can begin to transfer the actual data block. If there was a problem, the sender repeats the transfer of the header. The receiver will know that the header is being sent again because it would have received a NAK character from the previous attempt. Once the receiver has the header and the ACK character, it can concentrate on receiving the data block. As long as you use very different bit patterns for these two characters, it's unlikely that an ACK character could become a NAK character because of a data transfer error.


If it continues to encounter errors, the sender won't keep trying to send the header forever. The constant MAXTRY is set to tell the sender how many errors to count before aborting the attempt to send the current data block. The ACK and NAK characters are also used to confirm receipt of the data block. The feedback character is sent to the receiver only after the entire data block has been sent.

With this method, every type of communication error can be detected. This protocol eliminates the need for checksums, which is a common way of checking for data transfer errors in other communications software.

Reading key status

Once the data block is transferred, a final byte is sent to complete the process. But this byte is sent from the receiver to the sender. This also gives the receiver the ability to communicate an ESCAPE signal to the sender. However, this isn't really necessary because the receiver could simply exit the communications software with ESCAPE and then let the sender wait for a time out error.

Since this isn't the best solution, the protocol has the sender wait for an "escape byte" from the receiver after the data block has been completely transferred. If the sender receives a value of TRUE, it exits the program with an appropriate message. Otherwise, the sender continues with normal program execution.

Communicating this type of message must be allowed in both directions, since the sender could also decide to terminate the communication at any time. The procedure for this is different than with the receiver. When it starts, the SendABlock routine determines whether the  key has been pressed. If it has, it sends a special escape token instead of the actual data block header. This special escape token is known to the receiver. When the receiver recognizes this token, it considers it as a signal to exit the program.

By building this escape mechanism into the block protocol, we can avoid having to make a permanent escape query at a higher level. Both of the demo programs also deal with a file level above the block level. The file level uses the block level protocol to transfer entire files piece by piece. We won't go into detail here about the file level, since the program listings at the end of this section are well documented.

Remember that the routines at the byte and block level can be used to transfer any data between sender and receiver. Also, both the sender and receiver are able to abort the communication at any point. These routines could serve as the basis for your own data transfer programs, which can compete with commercial packages such as LapLink. They may not be quite as fast, however, because this would require all of the byte level routines to be written completely in assembly language.

The higher levels

If an error, such as a time out error at the byte level or the receipt of an escape token at the block level, occurs, this error should be communicated to the highest level as quickly as possible. The various levels (byte, block, and file) in these demo programs use procedures and functions to communicate with each other. Each time a routine ends with an error, it returns to the routine that called it, working its way back to the top level by level.

Modern C compilers support the `setjmp()` and `longjmp()` functions in these instances. These functions allow jumps across several program levels. The `setjmp()` function sets the location in the program code that will be the destination of the jump. If an error occurs, you can change program control to this location by using the `longjmp()` function. For more information about these functions, refer to your C compiler documentation.

Unfortunately, Turbo Pascal doesn't have similar functions. However, you can implement these commands yourself.

You'll find the following program(s) on the companion CD-ROM



PLINKP.PAS (Pascal listing)
PLINKPA.ASM (Assembler listing)
PLINKC.C (C listing)
PLINKCA.ASM (Assembler listing)



Keyboard Programming

The keyboard is one device that you are always concerned with in DOS programming. Since the keyboard is your computer's main input device, it's at the heart of most applications. TSR programs wouldn't even be possible without the keyboard.

TSR programs use the keyboard differently than normal applications. Usually, a program simply queries the keyboard to determine what keys the user has pressed. However, even this simple task can be filled with hidden complications. For example, the meaning of many keys are changed by the status of the **Caps Lock**, **Num Lock** and **Scroll Lock** keys. Other keys, such as the function keys and the cursor keys, change the meaning of a key press without changing the visible characters on the screen.

However, this can easily be managed, as we'll explain in the "Accessing The Keyboard From The BIOS" section. Unfortunately this process becomes more complicated with TSR and ISR routines that capture keyboard entry before the entry can reach the application. We'll explain how to do this in the third section, "The Keyboard Interrupt Handler".

In the "Programming The Keyboard Controller" section we'll show you how to program the keyboard directly. We'll also examine how a program can effect the way the keyboard functions. For example, you can change the key repeat speed or turn on and off the keyboard LEDs.

Keyboard Programming Basics

In Chapter 2 we examined the relationship between the hardware, DOS and the BIOS using the keyboard as our example. In this section we'll discuss this topic in more detail.

Keyboard to program

When the user presses a key on the keyboard, an electric impulse, which identifies the location of the key, is generated. This signal is handled by the keyboard processor, which is located inside the keyboard itself. Generally this processor is an Intel 8048 chip or an equivalent from another manufacturer. On AT class computers the communication is handled by an Intel 8042 chip. With this chip, ATs are capable of bi-directional communication between the keyboard and CPU. The earlier PC's and PC/XT's do not have this bi-directional communication capability.

Converting the scan code

The keyboard processor converts the electric impulse indicating the key position into a number called a *scan code*. There is no relationship between the scan code and the character printed on the key that was pressed or the function the key represents in the currently running program.

The keyboard processor passes the scan code to the computer. On an AT, the keyboard controller accepts the scan code. This transfer is done serially, since the cable that connects the keyboard and the computer has only one data line. This communication is synchronous, unlike the asynchronous communication found on a PC's serial port. Synchronous communication; is achieved by using a clock line and the data line. The clock line; transmits a timing signal by continuously switching from hi to lo (1 to 0). The transmission of the individual bits of the scan code are synchronized to this pulse.

If several keys are pressed simultaneously, the keyboard processor stores them in an internal buffer. The buffer usually has enough space for 10 keystrokes. However, you don't have to worry about this buffer becoming full because the data passes to the CPU much faster than a user can type.

Make and break codes

Scan codes are also generated when the user releases a key. This tells the system whether a key is still being pressed or has already been released. This is very important because it's the only way your computer can correctly interpret the situation when more than one key is pressed at once. Without this capability, you wouldn't be able to perform certain tasks, such as typing uppercase letters or rebooting your computer with **Ctrl+Alt+Del**.

Your system uses make codes and break codes; to distinguish the scan codes for keys that have been pressed ("make") and keys that have been released ("break"). The only difference is that bit 7 is set for a break code. This leads to two important consequences. First, break codes are always greater than 128 and make codes are always less than 128. Second, a PC keyboard cannot have more than 128 keys; otherwise the make codes would overlap the break codes.

The most obvious example of when more than one key must be pressed simultaneously is to type an uppercase letter. For example, to type an uppercase "A", the user presses and holds the right **Shift** key, then presses **A**. The keyboard controller passes the make code for the **Shift** key (36H), then the make code for the **A** key (1EH) to the computer. Since the system hasn't received a break code for the **Shift** key yet, it recognizes that both keys are being pressed simultaneously and generates an uppercase character instead of a lowercase one.

The ROM-BIOS keyboard handler

How does the processor receive these scan codes? The hardware interrupt IRQ1 is executed each time the keyboard sends a make code or a break code to the computer. This in turn calls interrupt 09H. This *keyboard handler* routine receives the make and break codes and converts them to the corresponding ASCII character codes, which can then be read by the application currently running (more on this later).

Many other tasks must be performed before the program can read the keyboard. First, the keyboard handler must read the make or break code from the keyboard using an I/O port. The address of this port is 60H for all PC systems. This port reads only make and break codes. The keyboard handler evaluates the codes and determines whether a character has been entered.

As we saw in our uppercase "A" example, not all keystrokes result in characters that are visible on the screen. The keyboard handler generated a character only after receiving the make code for the "A". A character wasn't generated when the make code for the **Shift** key was received. Think of entering the ASCII code for a character using the **Alt** key and the numeric keypad. In this instance, several keys are pressed and released before a character appears on screen.

Once the keyboard handler recognizes the character that was entered, it converts that character to a code the currently running application can understand. The scan codes themselves are unusable, because different keyboards use different sets of scan codes, although most sets are similar.

Scan code/ASCII code conversion

Therefore, scan codes are converted to ASCII codes, which are standard on all computers. Although the normal ASCII character set consists of 128 characters, PCs use an extended ASCII character set, which contains 256 characters. A listing of this character set can be found on the companion CD-ROM in the Appendices. The converted ASCII character isn't passed directly to the application. First it's stored in a buffer. The structure of this buffer and the way it works are described in the next section.

Now the keyboard handler has completed its work. The application can then read the characters from the keyboard buffer and process them. The ROM-BIOS interrupt 16H has several functions available for this purpose (refer to the "Accessing The Keyboard From BIOS" section later in this chapter for more information).

Using foreign language keyboards

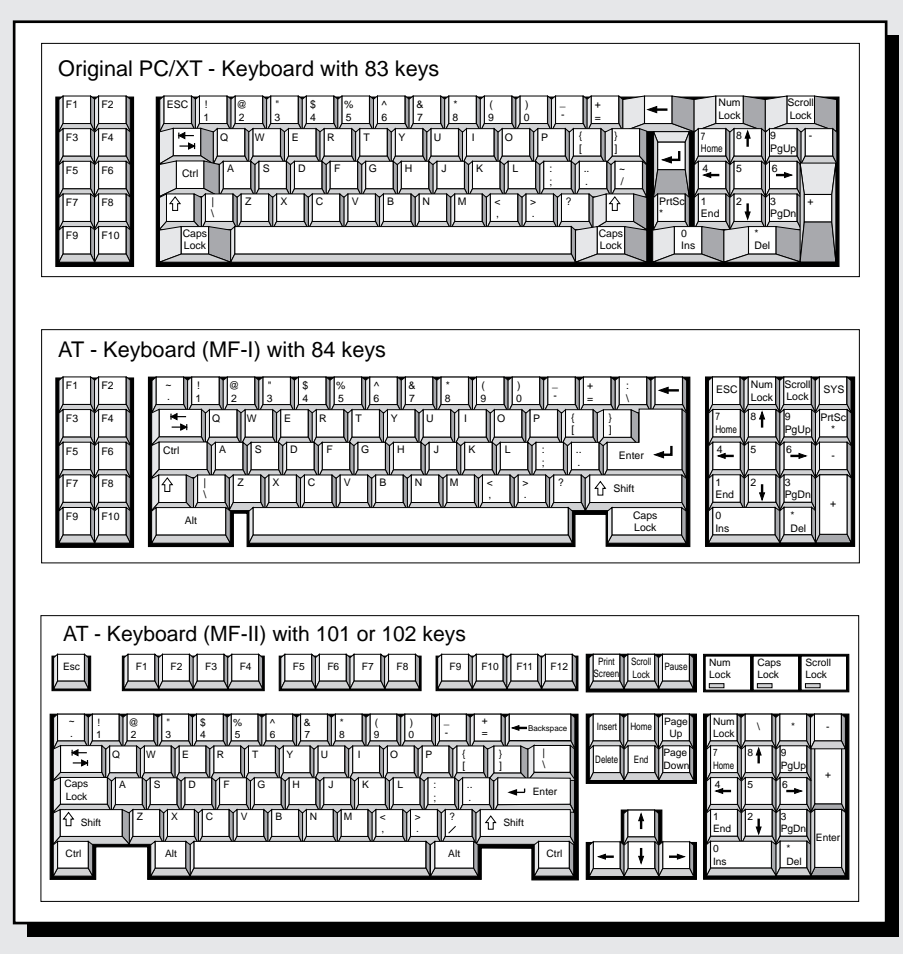
You should know one more thing about the keyboard handler. Although the ROM-BIOS defines which handler to use, DOS can replace this handler with another program. The handler in the ROM-BIOS is configured for the American keyboard by default. To use characters of other languages (such as Å, Ö, Ü, etc.), you can install another keyboard driver in your AUTOEXEC.BAT file. Installing another keyboard driver on your system will prove that, regardless of the characters printed

on your keyboard, the software converts the scan codes to ASCII codes, which determines what characters appear on the screen.

PC Keyboards

Various types of keyboards are available for PCs, but there are really only three standard keyboards. These keyboards were originally introduced by IBM. Although they are standard keyboards, their appearance can vary. This is because these keyboards are designed in many different languages, so the keys can be located in different places and contain different symbols. However, the number of keys and the scan codes they produce are standardized. Remember, the symbols printed on the keys don't always apply to what occurs inside the keyboard. The following illustration shows examples of the standard PC keyboards:

*Examples of
three standard
PC keyboards*



PC/XT and AT keyboards

The PC was first introduced with the PC/XT keyboard, which has 83 keys. The design of this keyboard has small **Enter** and **Shift** keys which were difficult to use. The AT keyboard, which has 84 keys, solved this problem. The **Enter** key and the two **Shift** keys were larger so you could easily find them, even when typing very quickly. However, since the keyboard itself wasn't larger, some other keys had to be smaller. The **Num Lock**, **Scroll Lock** and keypad **+** keys became smaller because they aren't used as frequently as the **Shift** keys.

The **[Sys Req]** key was also added to the keyboard. Although this key was intended to be used as a function key for calling operating system functions or TSR programs, developers never really adopted it.

The MF-II keyboard

The MF-II keyboard evolved from the MF-I keyboard that was developed for PCs and XTs but wasn't very popular. Many of the MF-II's features have become keyboard standards:

- A group of dedicated cursor keys that are separate from the numeric keypad.
- Function keys at the top of the keyboard.
- **[F11]** and **[F12]** function keys.
- **[Alt]** keys at the bottom of the keyboard for easier access.
- Three LEDs to indicate the status of the **[Num Lock]**, **[Caps Lock]** and **[Scroll Lock]** keys.

Two versions of the MF-II keyboard are available. The US version has 101 keys and the European version has 102. This allows an additional letter key to be added next to the left **[Shift]** key. Software (keyboard drivers in particular) can recognize an MF-II keyboard. The MF-II responds when checked with a corresponding identification code. The other two keyboard types don't have this capability.

Laptop and notebook keyboards

Various types of keyboards and keyboard layouts have appeared with the introduction of laptop and notebook computers. Although this can be very confusing, these keyboards usually emulate one of the three major PC standards we've described. This is accomplished either by emulating the standard scan codes within the keyboard or by converting nonstandard scan codes to standard ASCII codes with the keyboard handler. However, this solution isn't used frequently because it also requires changes to the keyboard driver.

Depending on licensing agreements, manufacturers can often modify keyboard drivers. However, the manufacturer may discover that many TSR programs won't work properly with their keyboard. This can occur because TSR programs generally read the keyboard at the lowest level, which is the scan codes themselves.

Keyboards and mouse emulation

Keyboards with mouse capabilities have recently become available. These include trackballs, special mouse pads or separate cursor keys that emulate mouse movements. As a programmer, you do not have to worry about any differences in these devices because they work with the standard mouse interface which we'll talk about in Chapter 11.

Accessing The Keyboard From The BIOS

Interrupt 16H provides three functions to read the keyboard and keyboard status. The BIOS keyboard functions are very limited. For example, there are no BIOS functions for removing characters from the keyboard buffer or renaming keys. DOS functions can perform these operations.

Interrupt 16H functions

The following functions are available to BIOS interrupt 16H:

Function 00H:	Read keyboard
---------------	---------------

Interrupt 16H usually receives a call when a program expects user input of one or more characters. If a character was already entered before the function call, this character is removed from the keyboard buffer and is passed to the calling program. If the keyboard buffer is empty, function 00H waits until a character is input and then returns to the calling program. The caller can determine the character or activate a key by examining the contents of the AL and the AH registers.

Control codes

As you already know, any ASCII code can be entered from the keyboard using the **[Alt]** key and the keys of the numeric keypad. However, it's also possible to use the **[Ctrl]** key. When used with other keys, this key can enter ASCII codes smaller than code number 32. The figure on the next page shows which keys can be accessed.

*Character input
with the **[Ctrl]** key*

Dec	Symbol	Keyboard codes	Dec	Symbol	Keyboard codes
0	Empty (Null)	[Ctrl] + [2]	16	▶	[Ctrl] + [P]
1	☺	[Ctrl] + [A]	17	◀	[Ctrl] + [Q]
2	☻	[Ctrl] + [B]	18	↕	[Ctrl] + [R]
3	♥	[Ctrl] + [C]	19	!!	[Ctrl] + [S]
4	♦	[Ctrl] + [D]	20	¶	[Ctrl] + [T]
5	♣	[Ctrl] + [E]	21	Ⓢ	[Ctrl] + [U]
6	♠	[Ctrl] + [F]	22	▬	[Ctrl] + [V]
7	●	[Ctrl] + [G]	23	↕	[Ctrl] + [W]
8	● BS	[Ctrl] + [H] [Shift] + [Bksp]	24	↑	[Ctrl] + [X]
9	○ TAB	[Ctrl] + [I]	25	↓	[Ctrl] + [Y]
10	● LF	[Ctrl] + [J] [Ctrl] + [Enter]	26	→	[Ctrl] + [Z]
11	○	[Ctrl] + [K]	27	← ESC	[Ctrl] + [Esc] [Esc] [Shift] + [Esc]
12	○	[Ctrl] + [L]	28	└	[Ctrl] + [V]
13	♪ CR	[Ctrl] + [M] [Shift] + [Enter]	29	↔	[Ctrl] + [J]
14	♪	[Ctrl] + [N]	30	▲	[Ctrl] + [6]
15	⊗	[Ctrl] + [O]	31	▼	[Ctrl] + [=]
			32	Space	[Spacebar] [Shift] + [Spacebar] [Ctrl] + [Spacebar] [Alt] + [Spacebar]

ASCII

If the AL register contains a value other than 00H, it contains the ASCII code of the character. The AH register contains the scan code of the active key. The code in the AL register corresponds to the ASCII codes for character output on the screen. Some differences occur in the control keys (see the table to the right).

ASCII codes 8, 9, 10, 13 and 27 have special meanings. This makes it difficult to use them as text characters. For example, the small left arrow character is ASCII code 27. However, most programs interpret ASCII code 27 as an escape command rather than a request to enter this character.

To avoid this problem in your programs, define a key, such as **F1**, that must be pressed before one of these special characters is entered. Then you must ensure that your program saves the last keystroke. So, if **Esc**, **Enter** or **Tab** is pressed, simply check to see whether the previous keystroke was **F1**. If it was, the keystroke should be interpreted as a text character instead of a control character.

ASCII control codes on PCs			
Code	Meaning	Code	Char.
8	Backspace	BS	
9	Tab	TAB	
10	Linefeed (Ctrl + Enter)	LF	XXX
13	Carriage Return	CR	
27	Escape	ESC	















































Extended keyboard codes

In addition to the ASCII codes, the BIOS functions also support the extended keyboard codes. The 256 characters of the PC ASCII character set include certain control characters, such as **Tab**, **Enter** and **Esc**, but not function keys and cursor keys.

So, the codes for these keys are returned according to a slightly different process, which utilizes ASCII code 0. The real information, the extended keyboard code, is found in the AH register. This is where you would normally find the scan code for the key.

If your program finds the value 00H in the AL register after calling function 00H or 01H of BIOS interrupt 16H, the keyboard code will be found in the AH register. With this method, an additional 256 character codes can be used. The following table lists the extended keyboard codes that can be read with functions 00H and 01H. Key combinations that aren't found in this table, such as all combinations of **Ctrl** + **Shift** + a letter key, aren't recognized by the BIOS and don't have their own keyboard codes.

Extended key codes			
Code (hex)	Code (dec)	Key(s)	
0FH	15	Shift Tab	
10H	16	Alt Q	(2nd keyboard series)
11H	17	Alt W	
12H	18	Alt E	
13H	19	Alt R	
14H	20	Alt T	
15H	21	Alt Y	
16H	22	Alt U	
17H	23	Alt I	
18H	24	Alt O	
19H	25	Alt P	
1EH	30	Alt A	(3rd keyboard series)

Extended key codes			
Code (hex)	Code (dec)	Key(s)	
1FH	31	 	
20H	32	 	
21H	33	 	
22H	34	 	
23H	35	 	
24H	36	 	
25H	37	 	
26H	38	 	
2CH	44	 	(4th keyboard series)
2DH	45	 	
2EH	46	 	
2FH	47	 	
30H	48	 	
31H	49	 	
32H	50	 	
03BH	59		
3CH	60		
3DH	61		
3EH	62		
3FH	63		
40H	64		
41H	65		
42H	66		
43H	67		
44H	68		
47H	71		
48H	72		
49H	73		
4BH	75		
4DH	77		
50H	80		
51H	81		
52H	82		
53H	83		
54H	84	 	
55H	85	 	

Extended key codes				
Code (hex)	Code (dec)	Key(s)		
56H	86	Shift	F3	
57H	87	Shift	F4	
58H	88	Shift	F5	
59H	89	Shift	F6	
5AH	90	Shift	F7	
5BH	91	Shift	F8	
5CH	92	Shift	F9	
5DH	93	Shift	F10	
5EH	94	Ctrl	F1	
5FH	95	Ctrl	F2	
60H	96	Ctrl	F3	
61H	97	Ctrl	F4	
62H	98	Ctrl	F5	
63H	99	Ctrl	F6	
64H	100	Ctrl	F7	
65H	101	Ctrl	F8	
66H	102	Ctrl	F9	
67H	103	Ctrl	F10	
68H	104	Alt	F1	
69H	105	Alt	F2	
6AH	106	Alt	F3	
6BH	107	Alt	F4	
6CH	108	Alt	F5	
6DH	109	Alt	F6	
6EH	110	Alt	F7	
6FH	111	Alt	F8	
70H	112	Alt	F9	
71H	113	Alt	F10	
73H	115	Ctrl	←	
74H	116	Ctrl	→	
75H	117	Ctrl	End	
76H	118	Alt		
77H	119	Alt	Home	
78H	120	Alt	1	(1st keyboard series)
79H	121	Alt	2	
7AH	122	Alt	3	

Extended key codes			
Code (hex)	Code (dec)	Key(s)	
7BH	123	Alt	4
7CH	124	Alt	5
7DH	125	Alt	6
7EH	126	Alt	7
7FH	127	Alt	8
80H	128	Alt	9
81H	129	Alt	0
82H	130	Alt	
83H	131	Alt	,

Keystroke combinations not included in this table cannot be read by the BIOS keyboard functions because they don't create keyboard codes. This applies to the function keys and to combinations of keys, such as **Ctrl**, **Alt** and **Shift**. Some DOS programs can interpret these keys because they have their own keyboard interrupt handlers that can be programmed to interpret any desired keystroke combinations.

BIOS-proof keys

Some key combinations cannot be read by BIOS as key codes because they execute commands. For example, activating the **Print** key or **Print Screen** calls BIOS interrupt 5H. This starts a routine that copies the current screen display to a printer, to produce a hardcopy. The **Ctrl**+**Num Lock** keys stop the system completely until the user presses another key. The keyboard buffer ignores the **Ctrl**+**Num Lock** keys and the next key pressed, so programs cannot read these keys.

Pressing the **Ctrl**+**Break** key combination calls interrupt 1BH. Usually the current program stops and returns to DOS. To prevent this from happening, direct this interrupt to a routine, within the application, that continues program execution even if this routine consists only of a single IRET assembly language instruction.

ATs and a few advanced PC/XTs contain the **Sys Req** key. When this key is pressed, interrupt 15H is called by passing the value 8500H to the AX register. When the user releases the key, the AX register then receives the value 8501H. The value 85H in the AH register represents the function number of interrupt 15H. When DOS is first started, function 85H of the BIOS interrupt 15H performs only an IRET instruction; pressing the **Sys Req** key has no visible result. To use this key in your program, you can include your own handler for interrupt 15H and route this function call to this handler.

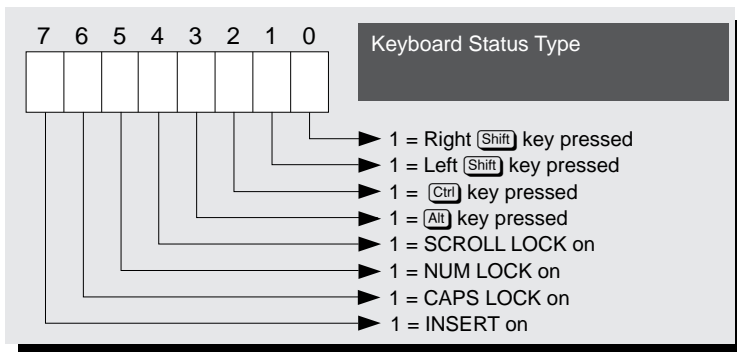
Function 01H:	Read keyboard
---------------	---------------

Function 01H also reads the keyboard. However, unlike function 00H, function 01H leaves the preceding character in the keyboard buffer. Repeated calls to function 01H or function 00H reread the keyboard. Place the value 01H in the AH register to call function 01H.

Unlike function 00H, after the function call, function 01H immediately informs the calling program, with the zero flag, whether a character is available. If the zero flag equals 1, a character isn't available. If the zero flag is 0, the AL and AH registers contain information about the activated key. As in function 00H, the AL register contains the value 00H if the user activated an extended key and a value unequal to 00H if the user pressed a "normal" key. The AH register contains the scan codes of normal keys; extended keys place their codes in the AH register.

Function 02H:	Read control keys
---------------	-------------------

Function 02H has a different task. This function reads the status of certain control keys and conditions (e.g., **Insert**). Place the number 02H in the AH register to call the function. The keyboard status can be found in the AL register after the function call.



For example, if bit 3 is set, the user is holding down the (Alt) key. If bit 6 is set, then the (Caps Lock) key is on and the user is typing in uppercase mode.

Notice the keyboard status byte is different depending on the left and right (Shift) keys, but not between the two different (Alt) and (Ctrl) keys of the MF-II keyboard. This is because both the keyboard status byte and the BIOS keyboard function 02H were developed with the first PCs, before the MF-II keyboard even existed.

The keyboard status byte is often used by TSR programs to detect when the hotkeys that activate the program are pressed. For more information on this subject refer to Chapter 35 which discusses TSR programming.

Sample programs

The following programs demonstrate the various functions of the BIOS keyboard interrupts we've discussed. The four programs can be divided into two groups. The first three programs are written in the higher level languages used throughout this book. They call the various functions of BIOS keyboard interrupts for their own uses. The fourth program is an assembly language program. It modifies the BIOS keyboard interrupt functions and processing and acts as a resident program that can be accessed at a keypress.

Checking key status

The higher level programs provide a subroutine or a function for reading characters from the keyboard. This by itself isn't special because these languages have their own instructions that perform the same task. The important feature of the function is that it accepts other tasks in addition to reading characters. It displays the status of the keyboard functions (Insert), (Caps Lock) and (Num Lock) in the upper-right corner of the screen. This is especially useful for XT and PC owners because most keyboards don't indicate the key status. AT keyboards and some XT keyboards provide light emitting diodes (LED) that indicate the status of these keys. Otherwise, you never really know whether the (Insert) or (Caps Lock) mode is on.

Each program begins with a routine that reads the status of the keyboard functions through function 02H of BIOS keyboard interrupt 16H. Since the program only uses the (Insert), (Caps Lock) and (Num Lock) modes, the program only views the three highest level bits in the keyboard status byte. Based on this status byte, a flag initializes for every keyboard function. These flags indicate the status of one of these functions or modes within the program. The status is reversed when compared with the current mode. For example, if the (Insert) mode is switched off, the flag corresponding changes to OFF. We'll explain this later.

Calling the interrupt function

After initializing the internal flags, the actual routine for keyboard reading can be called. This routine also uses function 2 of the BIOS keyboard interrupt to read the keyboard function status. Then it compares the current status of each individual function with the previous status stored in a flag. During its first call after the initialization routine, it determines whether the status of all three functions has changed since its previous status. The change in status causes the routine to display the new status on the screen.

This explains why the flag is reversed in the initialization routine. This enables the keyboard function status to be displayed on the screen during the first call the keyboard routines instead of after it changed by pressing a key.

Now the routine performs its actual task, which is reading the keyboard. It uses function 01H of the BIOS keyboard interrupt to detect whether a key is available in the keyboard buffer of BIOS. If a key isn't available, the program jumps to the beginning of the routine and reads the keyboard function status again. This creates a loop that runs until a keypress occurs. This loop ensures that any status change is documented immediately on the screen.

Reading the keys

If a character appears in the BIOS keyboard buffer, the loop terminates and BIOS keyboard interrupt function 02H reads the key. The last step of this routine tests for an extended key code. The program adds 256 to the code to inform the calling routine that an extended key code is received. Then control returns to the calling routine.

This routine reads characters from the keyboard and displays them on the screen. This process repeats until the user presses a certain key. If the user presses the **Num Lock**, **Caps Lock** or **Insert** key, the screen immediately displays the result.

This type of centralized keyboard routine can be used in other programs for additional tasks. For example, with the help of this routine, a macro conversion can change one key into a string of characters. Another application could display help text on the screen when the user presses a certain key. Lotus 1-2-3 and dBASE use this method for displaying help screens.

NOTE

A small problem occurs with keyboard flag output. Since displaying keyboard flags on the screen changes the cursor's position, subsequent screen output from the program occurs at unexpected locations. These can disrupt the screen display. To avoid this problem, the keyboard routine must determine the current cursor position before the keyboard flag display. Then the routine must restore the cursor position to its previous value after displaying keyboard status.

A similar problem occurs with the color. The flag output assumes a certain color and the original color must be restored after the output. The problem is that none of the three languages has a command to determine the current color. In Pascal programs for keyboard reading, only a special procedure can set the color by recording the colors in a variable and setting it with a command. With these variables, the keyboard routine restores the current color after the individual flags are displayed.

You'll find the following program(s) on the companion CD-ROM



KEYB.BAS (Basic listing)
KEYP.PAS (Pascal listing)
KEYC.C (C listing)

Reading MF-II keyboards

The **F11** and **F12** function keys aren't included in the listing of extended keyboard codes. There is a good reason for this because these keys cannot be read with functions 00H and 01H. The developers of the ROM-BIOS intentionally excluded these keys from being processed by functions 00H and 01H to maintain compatibility.

The keyboard handler writes the scan codes for these keys to the keyboard buffer, like any other keystroke. But functions 00H and 01H simply ignore them and proceed as if the keyboard buffer is empty.

New BIOS functions

Three new BIOS functions detect these codes. These functions were introduced with the AT and can now be found in almost every ROM-BIOS. Also, if these functions are excluded from the ROM-BIOS, DOS Versions 3.3 and up implement these functions with the DOS keyboard driver KEYB.

These new functions are assigned the numbers 10H, 11H and 12H. The way these functions are called and work is similar to functions 00H, 01H and 02H. Function 12H is slightly different from 02H in function results; we'll discuss this difference later.

The new keyboard codes

The differences between 00H and 01H and their counterparts 10H and 11H involve the codes that are returned. This applies almost exclusively to extended keyboard codes that represent keys that aren't found on the PC/XT and AT keyboards or key combinations (such as **Ctrl+Tab**, **Ctrl+↑** or **Alt+Esc**) that aren't supported by these keyboards.

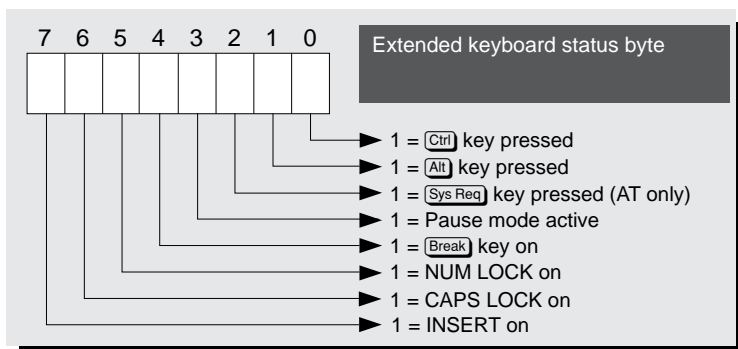
The following tables show the keys and key combinations that return different codes with functions 00H/01H and 10H/11H (all codes are in hexadecimal notation):

Extended key combinations when calling BIOS functions 10H/11H								
Function keys	Shift				Ctrl		Alt	
	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL
F11	----	85/00	----	87/00	----	89/00	----	8B/00
F12	----	86/00	----	88/00	----	8A/00	----	8C/00
Gray cursor keys in separate cursor block	Shift				Ctrl		Alt	
	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL
Home	47/00	47/E0	47/00	47/E0	77/00	77/E0	----	97/00
↑	48/00	48/E0	48/00	48/E0	----	8D/E0	----	98/00
Page Up	49/00	49/E0	49/00	49/E0	84/00	84/E0	----	99/00
←	4B/00	4B/E0	4B/00	4B/E0	73/00	73/E0	----	9B/00
→	4D/00	4D/E0	4D/00	4D/E0	74/00	74/E0	----	9D/00
End	4F/00	4F/E0	4F/00	4F/E0	75/00	75/E0	----	9F/00
↓	50/00	50/E0	50/00	50/E0	----	91/E0	----	A0/00
Page Down	51/00	51/E0	51/00	51/E0	76/00	76/E0	----	A1/00
Ins	52/00	52/E0	52/00	52/E0	----	92/E0	----	A2/00
Del	53/00	53/E0	53/00	53/E0	----	93/E0	----	A3/00
Other gray keys	Shift				Ctrl		Alt	
	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL
/	35/2F	E0/2F	----	E0/2F	----	95/00	----	A4/00
*	37/2A	37/2A	----	37/2A	----	96/00	----	37/00
-	4A/2D	4A/2D	----	4A/2D	----	8E/00	----	4A/00
+	4E/2B	4E/2B	----	4E/2B	----	90/00	---- 4E/00	
Enter	1C/0D	E0/0D	----	E0/0D	----	E0/0A	----	A6/00
Del	53/00	53/E0	53/00	53/E0	----	93/E0	----	A3/00

Extended key combinations when calling BIOS functions 10H/11H continued								
Additional combinations of white keys	[Shift]		[Ctrl]		[Alt]			
	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL	Old AH/AL	New AH/AL
[Tab]					----	94/00	----	A5/00
[5]					----	8F/00		
[↑]					----	8D/00		
[↓]					----	91/00		
[Ins]					----	92/00		
[Del]					----	93/00		
[Esc]							----	01/00
[Backspace]							----	0E/00
[Tab]							----	A5/00
[F1]							----	1A/00
[F2]							----	1B/00
[Enter]							----	1C/00
[;]							----	27/00
[']							----	28/00
[~]							----	29/00
[`]							----	2B/00
[.]							----	33/00
[/]							----	34/00
[~]							----	35/00

The codes for the gray cursor keys have been changed. This was done to distinguish them from the cursor keys of the numeric keypad. The gray cursor keys return ASCII code E0H, instead of 00H, to the AL register. When using BIOS functions 10H and 11H, you must have your programs look for ASCII code E0H as well as 00H to be able to read all extended keyboard codes.

If you're only interested in reading the [F11] and [F12] function keys and don't want to distinguish between the white and gray cursor keys, you can save yourself some work by changing ASCII code E0H to 00H after it's received. This enables you to handle both sets of cursor keys in the same way.



The new status functions

Functions 10H and 11H are identical to their original counterparts, but function 12H differs slightly from 02H in the returned result. In addition to the keyboard status byte, this function returns other information in the AH register. This extra information is known as the extended keyboard status byte. Although this is also a BIOS variable, it's only managed on systems with an MF-II keyboard.

Like the normal keyboard status byte, this byte also contains information on the current status of the various toggle keys. But it also provides the status of both the left and right **[Alt]** and **[Ctrl]** keys. The normal keyboard status byte doesn't allow you to read these separately.

Are these functions available?

As we mentioned, most BIOS manufacturers now include the extended BIOS keyboard functions or they are provided by DOS 3.3 and higher. However, you can't assume they will be available on all systems. Because of this, any program that wants to use these functions should first determine whether they are available. This isn't easy to do because no ROM-BIOS function can provide this information.

So you must use a trick. This consists of calling function 12H with the values 12H in the AH register and 00H in the AL register. If the value 1200H is found in the AX register after the function call, you can then be sure that function 12H and therefore the other extended BIOS functions, are unavailable on the system. This trick relies on an unwritten law of the ROM-BIOS, Under this "law", when an unknown function is called, the function call will end immediately and the contents of the AX register (the function and sub-function numbers) will be returned to the caller unchanged.

Sample programs

The MF2B.BAS, MF2P.PAS and MF2C.C sample programs will show how easily this test can be implemented in your program code. These implementations in BASIC, Pascal and C perform only one task; they read the keyboard using BIOS function 10H and display ASCII and scan codes entered at the keyboard. The output is in hexadecimal format so you can compare the values with the previous table.

These programs display the codes produced by the extra keys and key combinations available on the MF-II keyboard. This continues until the user presses the **[Esc]** key to end the program.

You can easily compare the extended BIOS functions with the normal functions. First, read a few codes with the extended function. Then switch the program to BIOS function 00H and press the same keys again. Simply change the line of code in the GetMFKey function where the BIOS function number is loaded in the AX register.

The keyboard query begins only after the TestMF function has determined that an MF keyboard is available. This involves the test previously described using extended BIOS function 12H.

The BIOS keyboard interrupt variables

The BIOS has eight variables in its variable segment for managing the keyboard and communication between the keyboard interrupt handler (interrupt 09H) and the BIOS keyboard functions (interrupt 16H). These are listed in the following table. These variables are useful for TSR programs that change the way the keyboard interrupts work. Even normal programs can find ways to manipulate these variables, as we'll see at the end of this section.

You should already be familiar with two of these variables: The keyboard status byte and the extended keyboard status byte. These two bytes are returned when calling functions 02H and 12H of interrupt 16H.

Offset address 19H is a byte used when entering ASCII codes with the **[Alt]** key and numeric keypad. When the number keys are pressed, the code entered is stored in this byte.

You'll find the following program(s) on the companion CD-ROM



MF2B.BAS (BASIC listing)
MF2P.PAS (Pascal listing)
MF2C.C (C listing)

BIOS variables for keyboard management		
Offset	Meaning	Type
17H	Keyboard status	1 byte
18H	Extended keyboard status	1 byte
19H	Code for ASCII input	1 byte
1AH	Next character in keyboard buffer	1 word
1CH	Last character in keyboard buffer	1 word
1EH	Keyboard buffer	16 words
80H	Start address of keyboard buffer	1 word
82H	End address of keyboard buffer	1 word

Managing the keyboard buffer

The three variables that follow at offset addresses 1AH, 1CH and 1EH manage the keyboard buffer. This is where the keyboard interrupt handler (interrupt 09H) stores keystrokes so applications can read them using the BIOS keyboard interrupt (16H). Before you can understand the significance of the first two variables, you must know the keyboard buffer is structured as a ring buffer. This kind of buffer is used when characters will be written to the buffer and read from it asynchronously (i.e., not within a specific time span). At first, it may seem that storing characters in sequence (the first character in the first position, the second character in the second position, etc.) is the best way to structure such a buffer. When a character is read, it's taken from the first position and all the other characters move up.

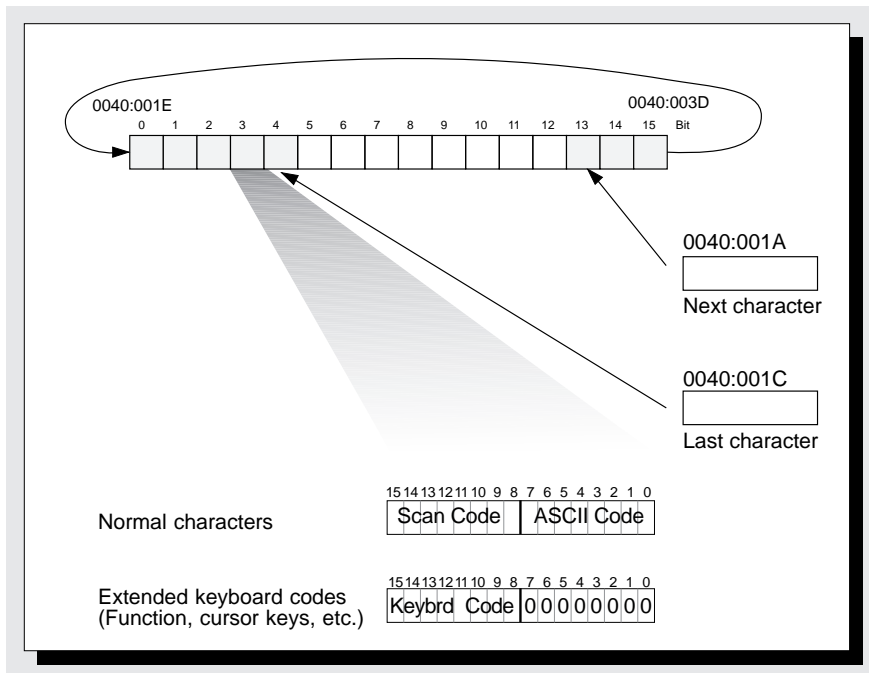
Although this method works, it generates unnecessary work for the processor because each time a character is read, all the characters in the buffer must be moved. So, the ring buffer uses two pointers. One pointer indicates the position from which the next character will be read and the other indicates the position where the next new character will be written.

With this method, the pointers move instead of the entire contents of the buffer. Initially, both pointers point to the beginning of the buffer. They move toward the end of the buffer with every read and write access. When the pointers reach the end of the buffer, they are reset to the beginning of the buffer. The ring buffer's name comes from this circular pointer movement. For example, suppose the offset address of the next character to be read from the keyboard buffer is 1AH. Once the character is read, the pointer moves two bytes towards the end of the buffer. Each character in the keyboard buffer requires two bytes: One byte for the ASCII code and one byte for the scan code. If the last location in the buffer was read, the pointer is set to the beginning of the buffer.

The same happens with the pointer at 1CH, which points to the position following the last character in the buffer. If the user presses another key, the key is stored at this location. The pointer then moves two bytes towards the end of the buffer. If the new keystroke was stored in the last word of the buffer, this pointer is reset to the beginning of the buffer. The relationship between the two pointers indicates the buffer's status. Two conditions are important:

1. If both pointers have the same value, this indicates an empty keyboard buffer.
2. If the end pointer tries to occupy the same space as the starting pointer, this indicates a full keyboard buffer.

Implementing the keyboard ring buffer and pointers



The BIOS keyboard buffer comprises 32 bytes. Since each character requires two bytes, the buffer can hold up to 16 characters. With a normal ASCII character, the ASCII code is stored first, followed by the scan code. For extended keyboard codes, the ASCII code will be 0 because the actual character code is in the subsequent byte.

Keyboard buffer location

The concept of the two ring buffer pointers suggests that this is a "mobile" buffer and that its location should be determined by the values of the pointers. However, the keyboard buffer is always located at offset address 1EH. Actually, this was the reason why the two ring buffer was designed; the two pointers would allow the size and location of the keyboard buffer to change as needed.

However, by the time these two variables were introduced with the modified BIOS of the first ATs, there were already numerous TSR programs that expected to find the keyboard buffer in a fixed location at offset address 1EH. So, the original idea never had a chance to be implemented.

Sample programs

The NOKEYP.BAS, NOKEYC.PAS and NOKEYB.C programs, which are in BASIC, Pascal and C, demonstrate how to manipulate the BIOS variables. These programs use a function that's very important to basic keyboard programming: Purging the contents of the keyboard buffer.

Occasionally you must purge the contents of the keyboard buffer. For example, if your program involves user prompts for deleting files or formatting disks, you don't want any accidental keystrokes stored in the keyboard buffer. The wrong key may tell your program to proceed before the user has had a chance to confirm the action.

Since the BIOS keyboard interrupt doesn't have a function for this, we must use a user-defined routine. Simply equalize the values of the two ring pointers, which creates the illusion the buffer is empty. It's also important to suppress all interrupts while doing this so new keystrokes aren't added to the buffer.

The NOKEYP.PAS and NOKEYC.C programs listed and stored on the companion CD-ROM in Pascal and C demonstrate how to do this. They begin with a countdown on the screen, which gives you time to enter some characters. Then the keyboard buffer is cleared and BIOS functions 00H and 01H are called to display all characters found in the keyboard buffer. After this, the keyboard buffer is purged and no characters exist.

BASIC implementation

Since interrupt suppression is more difficult in BASIC, in this version of the program a different method of clearing the keyboard buffer is used. This process calls BIOS functions 01H and 00H sequentially in a loop until the 01H function call indicates that no more characters are in the buffer.

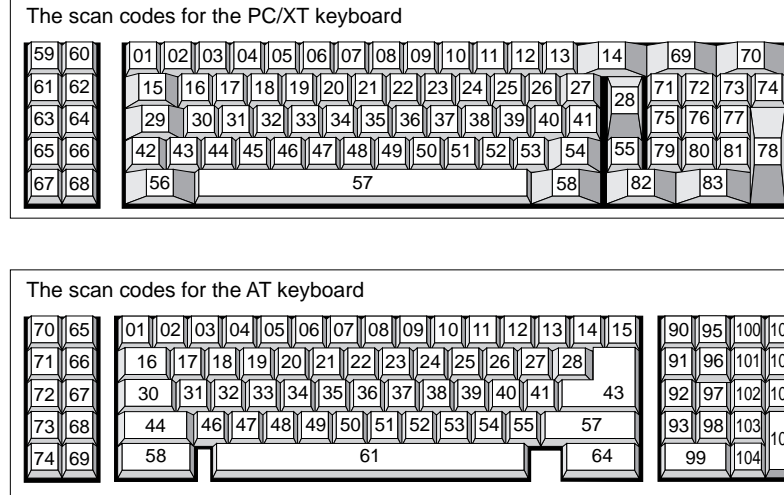
Scan codes

Although the ASCII character set and the extended keyboard codes are standardized, scan codes vary depending on the keyboard. All three keyboard standards work with a different set of scan codes. This is shown in the following illustration. It lists the scan codes for the PC/XT and AT keyboards.

You'll find the following program(s) on the companion CD-ROM



NOKEYP.PAS (Pascal listing)
NOKEYC.C (C listing)
NOKEYB.BAS (BASIC listing)

PC/XT and AT scan codes

You cannot always be sure the break code is the scan code plus 80H. For example, the AT keyboard sends two bytes when a key is released: F0H for a break code and then the key's scan code. However, the AT keyboard controller handles such incompatibilities. Anything sent through port 60H to the keyboard driver is converted to the normal format.

MF-II extended scan codes

Because of the AT keyboard controller, the MF-II keyboard also returns the same scan codes at port 60H as the AT keyboard, although they are slightly extended. The MF-II supports three scan code sets that are different from any previous keyboards.

The following table lists the additional MF-II keyboard scan codes. A byte containing the value E0H precedes each code. This byte indicates an extended scan code that must be handled in a special way. This is because most of the scan codes are already reserved and would otherwise be interpreted incorrectly.

In this instance, the break code equals the make code plus 80H and the initial E0H isn't changed. For the cursor keys of the gray cursor block that are used with the **(Shift)** key, the **(Shift)** key must first be suppressed. So, the break code for the **(Shift)** key is sent before this scan code.

For the left **(Shift)** key, the prefix of this make code is E0H AAH and the prefix of the break code is E0H D2H. For the right **(Shift)** key, the make code is E0H B6H and the break code is E0H D2H.

Extended Scan Codes on the MF-II Keyboard					
Gray cursor keys in separate cursor block			Function keys		
	Make	Break		Make	Break
(Home)	E0 47	E0 C7	(F11)	57	D7
(↑)	E0 48	E0 C8	(F12)	58	D8
(Page Up)	E0 49	E0 C9			
(←)	E0 4B	E0 CB			
(→)	E0 4D	E0 CD			
(End)	E0 47	E0 C7			
(↓)	E0 50	E0 D0			
(Page Down)	E0 51	E0 D1			
(Ins)	E0 52	E0 D2			
(Del)	E0 53	E0 D3			
All numbers are in hexadecimal notation					

You'll encounter a problem when programming with scan codes, for example in a TSR program. The only simple solution is to use an installation program in which the user actually press the hotkey combinations to be used. Then the program can store the appropriate scan codes. Unless absolutely necessary, don't program with scan codes. Instead, you should use ASCII codes or extended keyboard codes.

Although the scan codes generated by your keyboard or keyboard controller can be listed with the program described in the following section.

The Keyboard Interrupt Handler

The keyboard is the target of many special programs. Macro recorders, utilities for increasing the size of the keyboard buffer, special keyboard drivers and TSR programs activated by special hotkeys all affect the operation of the keyboard with their own interrupt handlers. Depending on the application, these interrupt handlers replace either BIOS keyboard interrupt 09H or 16H. This section contains examples of both types of programs (refer to Chapter 35) for detailed information about TSR programs).

Accessing BIOS keyboard interrupt 16H

If you want to extend or change the operations of one of the BIOS keyboard interrupts, simply redirect the interrupt to the interrupt handler within your program. Then when the interrupt executes, your own routine will be called. By doing this, you can insert new functions or redirect functions 00H and 01H to 10H and 11H. This is useful in all high level language programs, in which keyboard reading capabilities are based on various library routines.

Often, these routines simply use functions 00H and 01H of the BIOS keyboard interrupt, blocking the user from reading the **F11** and **F12** keys of the MF-II keyboard. But if you reroute these function calls to the extended functions that support MF-II, you can avoid having to write an extra routine to handle keyboard reading.

A macro utility

The MACROKEY.ASM program is a short macro utility written in assembly language. When you run the executable program from the system prompt, it remains resident in memory until you press the hotkey combination. This version of the program displays the text, "PC Intern published by Abacus" when you press **Alt+N**. You can easily modify the program to display other text by using any other hotkey combination or even by adding multiple macros.

A new BIOS keyboard interrupt handler

The heart of the MACROKEY.ASM program is a new interrupt handler for BIOS keyboard interrupt 16H. We'll discuss this new BIOS keyboard interrupt handler in detail. This handler, which is a routine called NEWI16, is located at the beginning of the program. At the start of this routine, the STI assembly language instruction enables hardware interrupts. (It's unnecessary to disable interrupts in this program.) The JMP instruction following the STI checks for an existing MACROKEY program in memory. The "MT" bytes indicate this.

NI1 reads the function number passed in the AH register. The program is only responsible for functions 00H, 01H, 10H and 11H. Functions 02H and 12H are sent to the old BIOS handler, whose address was stored when the program was started. So, the new handler modifies the old handler instead of completely replacing it.

If a call to one of the desired functions is discovered, the program execution branches to one of two locations, depending on the function required: Label FCT0 for 00H or 10H or FCT1 for 01H or 11H. The functions paired in each group can be handled identically because they both perform the same tasks. In either case, the program checks to see whether the macro has already been called.

If a macro hasn't been called, the new function must determine whether the macro hotkey has been pressed. The specified function is called for the old handler, which returns the function to the AX register if a key has been pressed. The key is compared with the main hotkey code (stored in the MKEY variable at the beginning of the program). The default hotkey is 3100H (**Alt+N**).

The new handler returns the result of the old handler to the caller if the hotkey isn't found. Notice the function call to 01H or 11H doesn't end with an IRET assembly language instruction. This would also retrieve the flag register from the stack, which was stored there as part of the calling INT instruction. This in turn would change the zero flag's contents, which indicates the availability of a key. The call ends with the VAR RET 2 instruction, which executes a FAR return to the caller just like IRET, but clears the flag register (2 bytes) from the stack without loading it.

If the hotkey is found, the program begins execution. All subsequent calls to 00H/10H or 01H/11H return a character from the macro buffer, instead of calling the old handler to get a keystroke from the keyboard buffer. This can be located using the MSTART variable, which can contain as many characters as you like. The MEND label must also be available. This indicates the offset address where macro playback ends. When this is reached, the old handler functions are called again and keystrokes are taken from the keyboard buffer.

The macro buffer contains only the ASCII character codes, not the scan codes. The value 0 is always returned as the scan code since most programs ignore this value.

Although this saves a lot of work when defining the macro text, it also makes using the extended keyboard codes in the macro text impossible. If necessary, you can always load both the ASCII code and the scan code in the macro buffer so both will be returned to the caller.

Redirecting keyboard hardware interrupts

If you want to access the keyboard at its lowest level, you must capture and redirect keyboard hardware interrupt 90H. We suggest using the existing handler since writing a completely new keyboard handler is a difficult task (refer to the KEYB.COM program using DEBUG).

Capturing scan codes

The GETSCAN.ASM program is a normal transient program that displays scan codes. It lets you view the scan codes of your keyboard on screen before they are actually routed to the keyboard handler. In this way, you can obtain the two byte make code sequences of the **F11** and **F12** function keys and the gray cursor keys, which are otherwise masked by the software.

Running the program in normal operating mode displays only make codes. If you start GETSCAN from the system prompt with the /R switch, break codes will also be displayed.

The main section of this program is the new interrupt handler for keyboard interrupt 09H, which executes each time the user presses or releases a key. The make or break code is received through port 60H and stored in the program's internal scan code buffer (the SCANBUF variable). It's managed as a ring buffer with two pointers, just like the BIOS keyboard buffer. These pointers are called SCANNEXT and SCANLAST. After the scan codes are stored, the program reverts to the old interrupt handler. The address of this handler was noted at the start of the program. This processes the keystrokes in the usual way and stores them in the keyboard buffer.

This is important because the keyboard buffer contents are read within the program using BIOS functions 01H and 00H until the user presses the **Enter** key. Once **Enter** is pressed, the program ends. Otherwise, the main program continues to read the internal scan code buffer between two BIOS calls. Whatever the new interrupt handler stores in the scan code buffer is immediately read by the main program and removed. The scan codes and break codes aren't lost until the program displays them.

This program enables you to read the scan codes that your keyboard uses and check for any differences from the standards. You may find some differences on the less expensive imported keyboards, but these differences usually involve seldom used key combinations.

You'll find the following program(s) on the companion CD-ROM



MACROKEY.ASM
(Assembler listing)

You'll find the following program(s) on the companion CD-ROM



GETSCAN.ASM
(Assembler listing)

Programming The Keyboard Controller

The keyboard is an independent unit in the PC system and has its own microprocessor and memory. The processor informs the system when a key is pressed or released. It does this by sending the system a scan code when a key is pressed or released. In both cases, the key is indicated by a code, which depends on the position of the key. These scan codes aren't related to the ASCII or extended keyboard codes to which the system later converts from the keypresses.

Communication with the system is performed over two bi-directional lines using a synchronous serial communications protocol. In addition to the actual data line used to transfer the individual bits, the clock line synchronizes the periodic transmission of signals. Transfers are made in one-byte increments, whereby a stop bit is transmitted first (with the value 0), followed by the eight data bits, beginning with the least significant bit. A parity bit, calculated using odd parity, follows the eighth data bit. Byte transfer then concludes with a stop bit, which forms the eleventh bit of the transfer. At both ends of the communications line (i.e., in the PC and in the keyboard itself) are devices that convert the signals on the data line to bytes and back again.

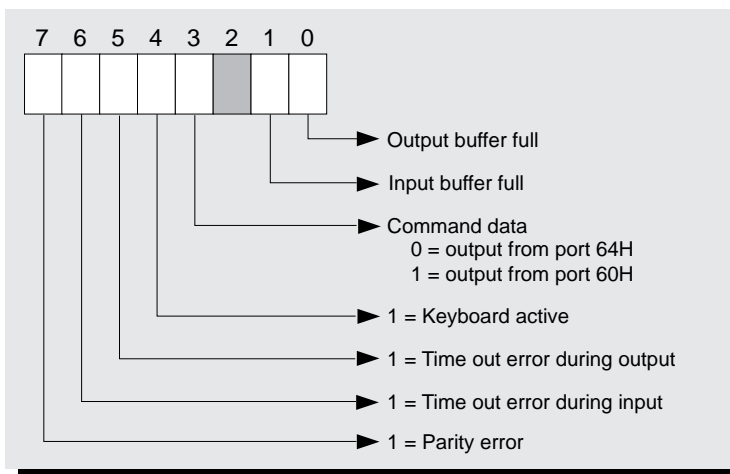
Although all types of PCs use this form of communication, we must distinguish between PC/XT and AT models. These systems use different processors as keyboard controllers. The Intel 8048 used in PC and XT keyboards is a relatively "dumb" device, which can only send scan codes to the system. However, the 8042 processor used in AT, 80386 and 80486 keyboards can do much more. With this processor, the communication between the system and the keyboard becomes more complex and the system can even control parts of the keyboard.

For the keyboard, the basis of this communication is represented by a status register and input buffer and output buffers. The buffers transfer the following:

- Keyboard codes that correspond to pressing or releasing a key.
- Data the system requests from the keyboard.

These buffers can be accessed at port 60H on the AT.

The input buffer can be written at port 60H as well as port 64H. The port that is used depends on the type of information to be transferred. If the system wants to send a command code to the keyboard, the code must be sent to port 60H, while the corresponding data byte must be sent to port 64H. Both end up in the keyboard input buffer, but a flag in the status register indicates whether a command byte (port 64H) or a data byte (port 60H) is involved.



In addition to this flag, bits 0 and 1 of the keyboard status register are especially important for communication with the keyboard. Bit 0 indicates the status of the output buffer. If this bit is 1, then the output buffer of the keyboard contains information that hasn't been read from port 60H yet. Reading from this port will automatically set this bit back to 0, indicating there is no longer a character in the output buffer.

Bit 1 of the status register is always set whenever the system has placed a character in the input buffer, before this character is processed by the keyboard. Nothing should be written to the keyboard input buffer unless this bit is equal to 0, which indicates the input buffer is empty.

Typematic rate

Of the various commands that a system can send to the keyboard, two are important to applications because they also play a role outside a keyboard interrupt handler. The first of these commands sets the typematic (repeat) rate of the keyboard. This is the number of make codes per second the keyboard will send to the system when a key is pressed and held down. It can be between two and 30 codes per second. To prevent the keys from repeating unintentionally, this repeat function doesn't begin until after a certain delay. This delay time can be set by the user and is encoded in binary as follows:

Coding for AT keyboard delay rate			
Code	Delay rate	Code	Delay rate
00b	1/4 second	10b	1/4 second
01b	1/2 second	11b	1 second

The keyboard will observe these times with a tolerance of 20%.

The repeat rate is also encoded in binary. The following table shows the relationship between the repeat (typematic) rate and the number of repetitions per second.

Typematic rate codes for the AT keyboard							
Code	RPS*	Code	RPS*	Code	RPS*	Code	RPS*
11111b	2.0	10111b	4.0	01111b	8.0	00111b	16.0
11110b	2.1	10110b	4.3	01110b	8.6	00110b	17.1
11101b	2.3	10101b	4.6	01101b	9.2	00101b	18.5
11100b	2.5	10100b	5.0	01100b	10.0	00100b	20.0
11011b	2.7	10011b	5.5	01011b	10.9	00011b	21.8
11010b	3.0	10010b	6.0	01010b	12.0	00010b	24.0
11001b	3.3	10001b	6.7	01001b	13.3	00001b	26.7
11000b	3.7	10000b	7.5	01000b	15.0	00000b	30.0

* Repetitions Per Second

This relationship may seem somewhat arbitrary at first, but it does follow a mathematical formula. The binary value of bits 0, 1 and 2 of the repeat rate form variable A and the binary value of bits 3 and 4 form variable B:

$$(8 + A) * 2B * 0.00417 * 1/\text{second}$$

The delay and repeat rate values are combined into a byte by placing the five bits of the repeat rate in front of the delay value. However, we can't simply send this value straight to the keyboard. First we must send the appropriate command code (34H) and then the repeat parameters. Both bytes must be sent to port 60H, but we cannot send them with an OUT instruction. We must use a transmission protocol that includes reading the keyboard status and which also accounts for the possibility the transfer might not work the first time. Since we must do this for both bytes, we should write a subroutine to do it. The structure of this subroutine is shown in the flowchart on the following page.

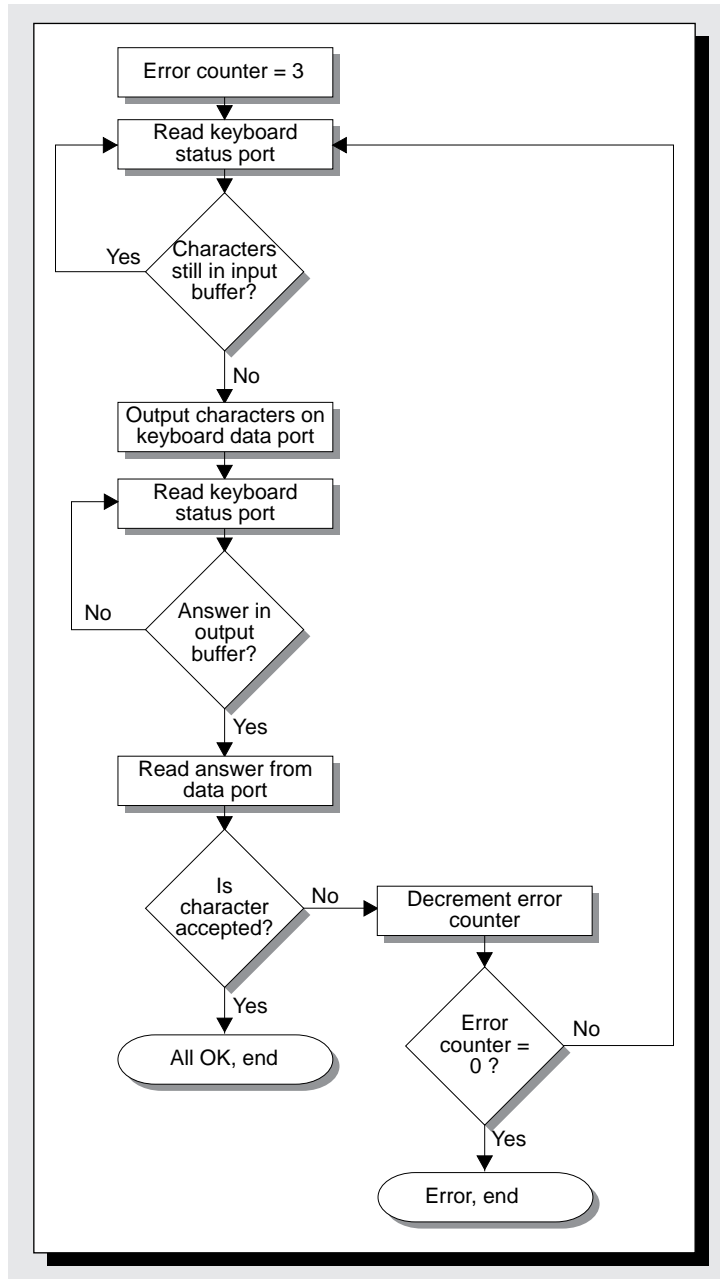
Sending bytes to the keyboard

First, we load an error counter that allows the routine to try to send the byte three times before an error is returned. Then the keyboard status port is read in a loop until bit 0 is cleared and the input buffer of the keyboard is empty. Then we can send the character to port 60H. To ensure the character was received (a parity error might have occurred, for example), the keyboard sends back a reply code. This has been received when bit 1 of the keyboard status port is set.

Programming steps for sending a byte to the keyboard

This register is read from port 64H in a loop until this condition is met. Now we can read the reply to our transmission from the keyboard data port. If it is the code 0FAH (acknowledge), the transmission was successful. Any other code indicates an error, which tells the subroutine to decrement the error counter and repeat the entire process, if the counter hasn't reached zero. In this case, the subroutine ends and signals an error to the caller.

*Program
flowchart-byte
transfer from
keyboard*



Sample programs

For an example of how this works, we've included the TYPMP.PAS, TYPMPA.ASM and TYPMC.C programs on the companion CD-ROM. They can be used to set the key repeat parameters on your keyboard. The heart of these programs is an assembly language routine that sends the parameters to the keyboard. This routine contains the subroutine we just discussed, which is first called to send the SetTym instruction to the keyboard. Another call is used to send the parameters themselves.

You'll find the following program(s) on the companion CD-ROM



TYPMPA.ASM (Assembler listing)
TYPMCA.ASM (Assembler listing)
TYPMP.PAS (Pascal listing)

The key repeat rate and the delay values are specified as separate parameters, following the program name entered at the DOS prompt. We also included the listing of the assembly routines for the various programs. The Pascal program includes these with `INLINE` statements; the linker links these statements to the C version of the program. To see the effect of the key repeat rate, first try setting the smallest repeat rate (0) and then the highest rate (30). Try pressing and holding a key at each of these settings to see the results.

LEDs

We can use this same method to switch the LEDs on the AT keyboard on and off. The corresponding instruction code is number 0EDH and is called the Set/Reset Mode Indicators instruction. After this command code has been successfully transmitted, the keyboard waits for a byte that reflects the status of the three LEDs. One bit in this byte represents one of the three LEDs, which is switched on when the corresponding bit is set. Setting and resetting these bits is useful only when the keyboard mode, which they indicate, is enabled or disabled.

Bit Number	LED
0	Scroll Lock
1	Num Lock
2	Caps Lock
3-7	Unused

These modes are managed in the BIOS instead of the keyboard. For example, the keyboard doesn't automatically convert all the letters to uppercase in `Caps Lock` mode. The keyboard can only associate a key with a virtual key number instead of a specific character. This key number is then converted to an ASCII or extended keyboard code by the BIOS. Obviously this also applies to the `Caps Lock` key, which simply sends a scan code to the computer when it's pressed. The BIOS assigns the `Caps Lock` function to this key by setting an internal flag that marks this mode as active, then sends the Set/Reset Mode Indicators instruction to the keyboard to switch on the appropriate LED.

Although these keyboard modes are usually enabled and disabled by the user pressing the corresponding keys, it may be useful to set a mode from within a program. This applies to keyboards that have separate cursor keys and a numerical keypad, for example. Since most keyboards can only enter numbers when Num Lock mode is on, it makes sense to set this mode automatically when the system is started. To do this we simply set the appropriate BIOS flag and then switch on the corresponding LED on the keyboard to inform the user that this mode has been activated.

In practice, a program simply must set the appropriate BIOS mode, since the BIOS automatically controls the keyboard LEDs. Whenever one of the functions of the BIOS keyboard interrupt is called, the BIOS checks to see whether the status of the LEDs matches the keyboard status, as indicated in an internal variable. If an inconsistency is found, the BIOS automatically sets the LEDs to the status given in the keyboard status flag.

Since the position of this flag in the BIOS variable segment and the meaning of the individual bits is completely documented (see Chapter 3), we can easily change these modes.

LEDB.BAS, LEDP.PAS and LEDC.C have routines to enable or disable the individual modes. Although PCs and XTs have corresponding LEDs, these programs won't work or change the modes without changing the status of the LEDs on a PC or XT keyboard. This is because these keyboards are equipped with an 8048 processor, which doesn't offer the ability to manage the LEDs. The fact these LEDs do switch on and off according to the modes isn't related to the BIOS and is handled directly by the keyboard.

You'll find the following program(s) on the companion CD-ROM



LEDB.BAS (BASIC listing)
LEDP.PAS (Pascal listing)
LEDC.C (C listing)



Joysticks

Thanks to the recent improvements in video cards and joysticks, PC versions of many video games have become popular. In this chapter you'll learn how joysticks are connected to the PC, how to "read" the joystick from your own programs, and how the BIOS can help you adapt programs for joysticks.

Three demonstration programs for this chapter (which you'll find on the companion CD-ROM) show how joystick reading works in BASIC, Pascal, and C.

Connecting joysticks

The PC's hardware usually isn't configured to accept joysticks. An additional expansion card provides the hardware needed to implement joystick control. This card, called a *game control adapter* or simply a *game card*, is usually a half-size expansion card that plugs into one of the PC's expansion slots. Although these cards usually contain two connectors for two joysticks, usually you'll need only one.

The PC uses analog joysticks. This means that you can't simply use the joysticks from your old game machine or home computer and plug them into the PC; those joysticks are digital. The analog joystick handles a variety of information that isn't sensed by the digital joysticks, such as intensity. However, you'll see that this feature can sometimes be a disadvantage.

PC joysticks are equipped with two buttons that can be read independently of one another, and can be utilized to fit the software application. A few PC joysticks even have special purpose buttons (e.g., for rapid fire).

Reading Joysticks From BIOS

You don't need software drivers to monitor the joysticks because both the hardware and software interfaces were defined in the early stages of PC design. Hardware ports 200H to 20FH are reserved for a game card. The BIOS refers to the default values provided by these ports.

Two functions are available for software interfacing. These functions (subfunctions 00H and 01H) can be accessed from interrupt 15H, function 84H. They execute a joystick check using the polling method, which takes the current joystick position and joystick button status directly from the hardware ports.

However, the joysticks can be monitored only with the polling method because a special hardware interrupt, which executes when the user moves the joystick or presses a button, is assigned to the joystick cards. So, a program equipped for joystick access continually depends on subfunctions 00H and 01H for joystick movement.

Determining the joystick position

The joystick position can be determined by placing 84H in the AH register and 01H in the DX register, which calls function 84H, subfunction 01H. If the carry flag contains 0 after calling interrupt 15H, the BIOS supports the function (i.e., a game port exists), and joystick position data can be found in the AX, BX, CX, and DX registers.

The first joystick's position is conveyed by the contents of the AX and BX registers (indicating X-position and Y-position, respectively). The second joystick's position is indicated by the contents of the CX and DX registers (again, indicating X-position and Y-position). The contents of these registers also indicate whether a joystick is connected to the port. If a stick is connected, the corresponding registers return values other than 0. So, if you find the CX and DX registers contain 0 after executing the function, a second joystick is connected. This rule also applies to the first joystick: If the AX and BX registers return 0, there isn't a first joystick.

Values other than 0 represent the joystick's position along the corresponding axis. However, there isn't a standard for these values. The values depend on the *potentiometers* (variable resistors, similar to the volume knob on a radio or television) contained in the joystick, which convert the joystick's position into a voltage. Different joysticks from different manufacturers and even different joystick models made by the same manufacturer return different voltages.

Many joystick oriented programs begin by prompting the user to move the joystick to the upper-right and the lower-left corners of the screen. This helps determine the range of movement offered by the potentiometers. Our in-house joysticks offer a range from 10 to 120 in the X-position, and from 9 to 102 in the Y-position. However, other joysticks may produce different values. It's important to remember that X-position values ascend from left to right, and the Y-position values ascend from top to bottom (not from bottom to top).

Though it would be easy to assume the joystick movement could be transformed into a linear coordinate system after simply determining its value range, this isn't true. The potentiometers in most joysticks operate exponentially rather than linearly.

Exponential operation means the center position of the joystick doesn't correspond to the median value of the value range. The demonstration programs at the end of this chapter will demonstrate this as you run them and test your joystick.

To avoid the problems that result from the exponential measurements of the potentiometers, many programs avoid implementing joystick coordinates in a linear coordinate system. For example, a game such as PacMan needs to recognize only four directions for game object movement (up, down, left, and right).

This type of application checks the joystick's position at the beginning of the program (i.e., the center position). From there, the application needs to compare only the center position with the current joystick values to determine the direction of movement. However, you cannot determine the intensity of the movement by using this method.

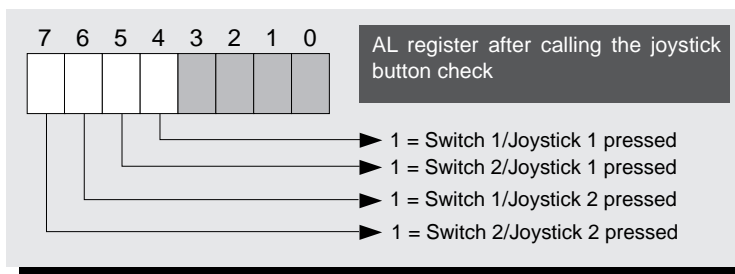
Reading the joystick buttons

In addition to the joystick position, the two joystick buttons are important to program operation.

The joystick position can be determined by placing 84H in the AH register and 00H in the DX register, thus calling function 84H, subfunction 00H. If the carry flag contains 0 after calling interrupt 15H, the BIOS supports the function (i.e., a game port exists), and the bits of the AL register indicate the status of the four available buttons on a set of two joysticks. The button being pressed sets the corresponding bit to a value of 1.

If the joystick buttons are arranged one above the other, the top button will be considered the first, and the lower button will be considered the second. When the joystick buttons are arranged side by side, the left one is considered the first, and the right one the second.

The following illustration shows the values of the AL register:



When working with this function, remember that it doesn't have a buffer function. So, it displays only the current joystick button status. If the user briefly presses a button when a joystick button check isn't being performed, the program won't sense the input and won't react to the user's action. So, if your program intends to use these buttons, it's especially important to permanently monitor the joystick button status in your programs using subfunction 00H.

Sample Programs

The three sample programs in BASIC, Pascal, and C read joystick position and button status using the GetJoyPos and GetJoyButton functions. Within the main program, these functions are used to first determine the joystick's minimum and maximum positions.

The program prompts the user to press the joystick up and right, and press a joystick button while the stick is in this position. This informs the program which joystick is being used (if two joysticks exist).

The program then prompts the user to press the joystick down and left, and press a joystick button while the stick is in this position. The program converts this range into the 80 columns and 25 rows of the text screen, and the current joystick position is calculated and indicated by an uppercase X. The resistance values yielded by the potentiometers are displayed in the upper-left corner of the screen, so the user can see how the joystick operates.

The program ends when the user presses both joystick buttons simultaneously.

You'll find the following program(s) on the companion CD-ROM



JOYSTB.BAS (BASIC listing)
JOYSTP.PAS (Pascal listing)
JOYSTC.C (C listing)

11

Mouse Programming

Using a mouse for PC applications was considered a luxury a few years ago. Today, however, most applications require that you use a mouse. One reason for the mouse's popularity is the development of new and more powerful video standards such as EGA, VGA, and Super VGA. These video cards helped advance graphical user interfaces, such as Microsoft Windows. These interfaces are almost unusable without a mouse.

Applications and operating systems benefit from mouse support. Ventura Publisher and Microsoft Works both use the mouse extensively. Also, DOS Version 4.0 accepts mouse as well as keyboard input.

A software interface acts as the connection between a program and the mouse. Although Microsoft Corporation designed this interface for its own mice, other mouse manufacturers accept this interface as a standard. The interface was made available to the industry as a minimum standard to retain compatibility with the Microsoft mouse.

This function interface is usually installed either through a device driver that is loaded during system boot or through a terminate and stay resident (TSR) program, such as MOUSE.COM, that's included with the Microsoft mouse package.

Mouse Functions

Mouse functions can be accessed in the same way as DOS and BIOS functions (refer to Chapters 14 and 8 for more information on the techniques used for addressing DOS and BIOS functions). The individual functions can be called through interrupt 33H. The identification number of the function must be passed to the AX register. The other processor registers are used in various combinations to pass information to a function.

Although 53 different functions can be called in this way, most applications use only a few of these functions. Before we examine each function, let's look at the concepts behind the mouse interface. This will help you understand how the individual functions work. In our discussion, we deliberately concentrated on text oriented mouse control. Pixel oriented applications should use a graphical interface, such as the Windows API, because this interface provides friendlier functions for mouse input than the programming interface found in this chapter.

In the following table we've listed the functions found on mouse drivers up to and including Version 8.0. You'll find more than enough functions needed for mouse control in a text oriented application.

Function	Task	Version
00H*	Reset mouse driver	01
01H*	Display mouse cursor	01
02H*	Hide mouse cursor	01
03H*	Get cursor position/button status	01
04H*	Move mouse cursor	01
05H*	Determine number of times mouse button was activated	01
06H*	Determine number of times mouse button was released	01
07H*	Set horizontal range of movement	01
08H*	Set vertical range of movement	01

Function	Task	Version
09H	Set mouse cursor (graphic mode)	01
0AH*	Set mouse cursor (text mode)	01
0BH*	Determine movement values	01
0CH*	Set event handler	01
0DH	Enable light pen emulation	01
0EH	Disable light pen emulation	01
0FH	Set cursor speed	01
10H*	Exclusion area	01
11H	Undocumented	01
12H	Undocumented	01
13H*	Set maximum for mouse speed doubling	01
14H	Exchange event handlers	01
15H	Determine mouse status buffer size	01
16H*	Store mouse status	01
17H*	Restore mouse status	01
18H*	Install alternate event handler	01
19H	Determine address of alternate event handler	01
1AH	Set mouse sensitivity	01
1BH	Determine mouse sensitivity	01
1CH	Set mouse hardware interrupt rate	01
1DH*	Set display page	01
1EH*	Determine display page	01
1FH	Disable mouse driver	01
20H	Enable mouse driver	01
21H	Reset mouse driver	01
22H	Set language for messages	01
23H	Get language number	01
24H	Determine mouse type	01
25H	Get general driver information	06
26H	Get maximum virtual coordinates	06
27H*	Get masks and mickey counts	7A
28H	Set video mode	07
29H	Count video modes	07
2AH	Get cursor hotspot	7B
2BH	Set acceleration curves	07
2CH	Read acceleration curves	07
2DH	Set/get active acceleration curves	07
2EH	Undocumented	01

Function	Task	Version
2FH	Mouse hardware reset	7B
30H	Set/get ballpoint information	7C
31H	Get minimum/maximum virtual coordinates	7D
32H	Get active advanced functions	7D
33H	Get switch settings	7D
34H	Get MOUSE.INI location	08
Legend: * = Commonly used function (detailed in this chapter) 01 = Version 1.0 and up 7B = Version 7.02 and up 06 = Version 6.26 and up 7C = Version 7.04 and up 07 = Version 7.0 and up 7D = Version 7.05 and up 7A = Version 7.01 and up 08 = Version 8.0 and up		

About mouse buttons

Unlike the keyboard, which has many keys and keyboard codes for each key, a PC mouse usually has two or even three mouse buttons. These buttons enable the user to select data in an application program.

The mouse buttons are also used to move the mouse cursor on the screen. The actual position of the mouse cursor on the screen is important. The mouse driver software always interprets the cursor's location on the screen relative to a virtual graphic screen. This virtual screen's resolution depends on the video mode and video card currently being used.

Since this virtual graphic display screen is also used within the text modes to determine the mouse's position and forms the basis for communication with the mouse interface, a conversion occurs between the graphic coordinates and the mouse cursor's line/column position. Since every column or line corresponds to eight pixels, the graphic coordinates must be either be divided by eight or shifted three places to the left in binary mode, which mathematically produces the same result. However, the processor performs the shifting much faster than it can perform the actual division.

About the mouse cursor

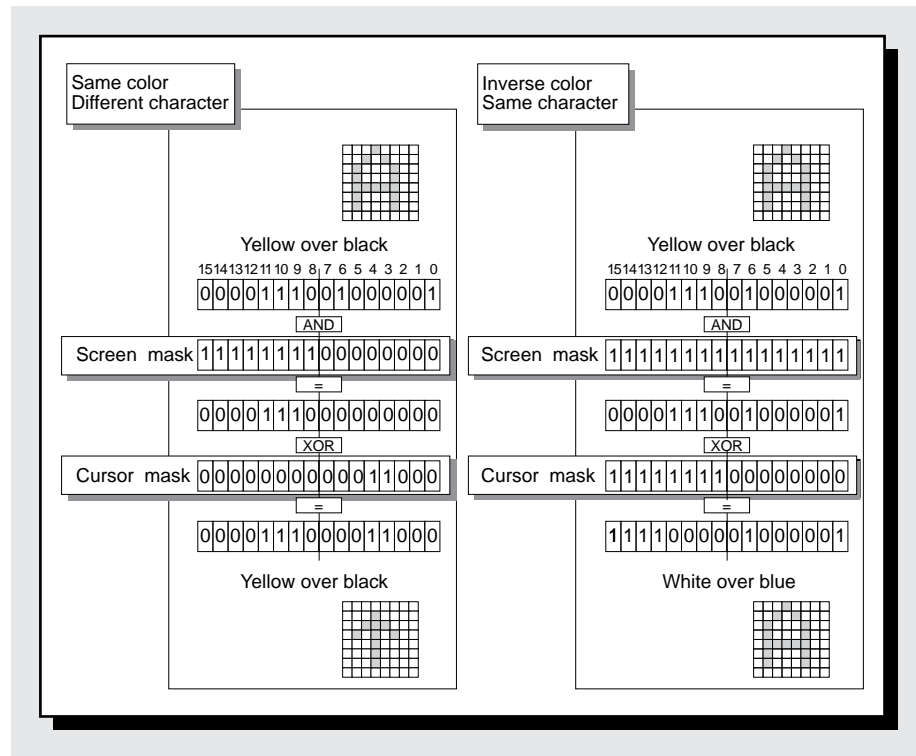
The mouse cursor shows the mouse's relative location on the screen. The cursor's shape can vary depending on the application and it can even change its appearance within an application. Word processors often display the mouse cursor as a block, similar to the text cursor. In text mode, the application can only determine the starting and ending line of the cursor. The cursor's size depends on the current character matrix and video mode. The options for creating a software cursor are more complicated because two 16-bit values, called the screen mask and cursor mask determine the cursor's appearance.

The mouse driver must determine the appearance of the cursor each time the cursor's position on the screen changes. The cursor mask and screen mask values are linked with the two bytes that describe the character code and the character color within video RAM. This linkage occurs in two steps. First, the character code and the attribute byte are combined with screen mask through a binary AND. The result of this connection is then combined with the cursor mask through an exclusive OR. The result then appears on the screen.

This type of combination provides several options for changing the cursor's appearance. Four of the most common cursor options are:

- Pointer appears as one specific character in one specific color.
- Pointer appears as one specific character, but the color changes when the cursor overlaps a character (e.g., inverse video).
- Pointer appears as one specific character, but the character color changes when the cursor overlaps a character.
- Pointer appears as one specific character, but character color changes to a variant of the character color when the cursor overlaps a character.

Formation of mouse cursor by combination of current character, cursor and screen mask



The standard measurement unit in the mouse interface is the *mickey*, which is named after Mickey Mouse. Originally one mickey equaled 1/200". This measurement applies to older systems, which have a resolution of 200 pixels per inch. Newer mice and video cards have mickey measurements of 400 mickeys per inch. Although the mouse driver compensates for additional pixel information, we'll use this as the measurement standard in this chapter.

Function 00H:	Reset mouse driver
---------------	--------------------

A program should call the function 00H before calling any of the mouse functions. This function resets the mouse driver. It can also determine whether a mouse and mouse driver exist by examining the content of the AX register after the function call. If the AX register contains the value 0000H after the function call, a mouse driver wasn't installed. Even if a mouse is connected, the mouse driver no longer exists. If a mouse driver and mouse exist, function 00H returns the value FFFFH in the AX register. The BX register contains the number of buttons on the mouse. As we mentioned, PC mice usually have two mouse buttons, although some mice have three buttons. Since very few applications need or use three buttons, usually you'll only need two buttons.

Function 00H resets the numerous mouse parameters to their default values. The mouse cursor moves to the center of the screen. The cursor mask and screen mask are defined so the cursor appears as an inverse video rectangle. Video page 0 is selected as the default page on which the cursor appears. The cursor disappears from the screen immediately.

Function 01H:	Display mouse cursor
---------------	----------------------

Function 01H displays the cursor on the screen. Load the function number into the AX register; additional parameters aren't needed. Since the mouse driver follows the movement of the mouse even when the mouse cursor has been disabled, the mouse cursor may not reappear at the position where it was when it disappeared.

Function 02H:	Remove mouse cursor
---------------	---------------------

Function 02H removes the mouse cursor from the screen. Load the function number into the AX register; additional parameters aren't required. The calls between functions 01H and 02H must be performed in the proper sequence to be effective. For example, calling function 02H twice in succession means that you must also call function 01H twice in succession to return the cursor to the screen.

Functions 01H and 02H aren't used very often. Usually, you simply must call function 00H and function 01H at the beginning of a program and call function 02H at the end of the program. These functions are frequently used when the application program writes characters directly into video RAM, bypassing the slow DOS and BIOS display routines. Avoid writing characters over the mouse cursor; otherwise the following will occur:

1. The mouse cursor disappears if overwritten by another character.
2. The mouse driver produces the wrong character on the screen when the user moves the mouse cursor. Before the cursor appears at a certain position on the screen, it records the character that occupied this position until now. This character is restored to the old position as soon as the cursor moves to another position on the screen. During a direct write access to video RAM, the driver doesn't record that a new character was output at the position of the cursor. So, the old (and incorrect) character is displayed on the screen while the cursor is moved.

To prevent this, remove the cursor before character output and return the old character to the screen. The new character will be stored when the cursor is restored to the screen. However, don't do this for every character output because it slows the system down and negates the advantages of direct access to video RAM. We recommend removing the cursor once from the screen before extensive output, such as construction of a screen window. After the operation, the cursor can be restored on the screen.

Although the DOS and BIOS character output functions write their output directly to video RAM, don't worry about programming the cursor when working with these functions. During installation, the mouse driver moved interrupt vector 10H, which handles BIOS and DOS screen output, to its own routine. So, the driver can then display or disable the cursor as needed.

Function 04H:	Move mouse cursor
---------------	-------------------

With function 04H you can move the mouse cursor to a specific location on the screen without moving the mouse. Pass the function number to the AX register, the new horizontal coordinate (column) to the CX register, and the new vertical coordinate (line) to the DX register. Remember these coordinates, like all other functions, must be relative to the virtual screen. Text coordinates must be multiplied by eight (or shifted left three binary places) before they can be passed to function 04H. The coordinates must be located inside a screen area designated as the mouse's range of movement.

Function 00H sets the complete range of the mouse's movement to the entire screen area. Functions 07H and 08H limit this range to a smaller area.

Function 07H & 08H:	Set range of movement
---------------------	-----------------------

Function 07H specifies the horizontal range of movement. Pass the function number to the AX register, the minimum X-coordinate to the CX register, and the maximum X-coordinate to the DX register. Function 08H specifies the vertical range of movement. Pass the function number to the AX register, the minimum Y-coordinate to the CX register and the maximum Y-coordinate to the DX register.

After calling these functions, the mouse driver automatically moves the cursor within the range, unless it's already within the indicated borders. The user cannot move the cursor outside this range.

Function 10H:	Exclusion area
---------------	----------------

Besides the area of movement allotted to the cursor, the mouse driver also provides an exclusion area. When the user moves the cursor into this section of the screen, the mouse cursor becomes invisible. The mouse cursor becomes visible again as soon as the user moves the cursor away from the exclusion area. This area is undefined after the call of function 00H. It can be

defined at any time by calling function 10H, but the mouse driver can control only one exclusion area at a time. The coordinates of the exclusion area are passed to function 10H in the CX:DX and SI:DI register pairs. These register pairs specify the upper-left corner and lower-right corner respectively. CX and SI accept the X-coordinate, and DX and DI accept the Y-coordinate.

The exclusion area and function 02H perform special tasks during direct access to video RAM. Although function 02H removes the cursor from the screen, this can occur in conjunction with function 10H only if the cursor is already within the exclusion area, or if the user moves the cursor within the exclusion area. This makes function 10H useful when creating large display areas (e.g., a window). This allows the cursor to remain on the screen as long as it isn't within this exclusion area.

The exclusion area can be removed by calling function 01H or function 00H. Function 01H makes the cursor visible automatically if it's already within the exclusion area.

Function 1DH:	Set display page
---------------	------------------

Function 1DH sets the display page on which the cursor appears. This function is needed only if the program switches a display page other than the current one to the foreground through direct video card programming. Pass the number of the display page to the BX register. When BIOS interrupt 10H activates a display page, this function can be omitted because the mouse driver will automatically adapt to the change.

Function 0FH:	Set cursor speed
---------------	------------------

Two parameters determine the speed at which the mouse cursor moves on the screen. These parameters specify the relationship between the distance of a cursor movement and the pixels traversed in the virtual mouse display screen. Function 0FH allows the user to set these parameters for horizontal and vertical movement. The parameters are passed in the CX and DX registers (horizontal and vertical, respectively). These numbers indicate the number of mickeys, which correspond to eight pixels in the virtual mouse display screen. These pixels correspond to one line or column in the text mode display screen.

The default values after calling function 00H are 8 horizontal mickeys and 16 vertical mickeys. In text mode, the cursor moves one column after the cursor is moved 8 mickeys horizontally. A jump to the next line occurs only after the cursor is moved 16 mickeys vertically.

Usually these settings can be used as default values because they work with all resolutions in text mode. This function also enables you to make the cursor movement faster or slower.

Function 0AH:	Set cursor shape
---------------	------------------

Function 0AH determines the appearance of the cursor in text mode. The cursor mask and screen mask mentioned earlier are determining factors of the cursor's appearance in text mode. Pass 0AH to the AX register and the value determining the cursor's shape to the BX register.

Software-specific cursor

If the BX register contains the value 0, the mouse driver selects the cursor as specified by the software. The screen mask number must be loaded into the CX register and the cursor mask number must be loaded into the DX register. These numbers indicate the addresses from which the mouse driver can access cursor shape parameters.

Hardware-specific cursor

If the BX register contains the value 1, the mouse driver selects the cursor as specified by the hardware. The starting line of the hardware cursor must be loaded into the CX register, and the ending line must be loaded into the DX register.

Video mode and cursor size

Remember the allowable values for the starting line and ending line depend on the video mode currently in use:

- The monochrome display adapter accepts values from 0 to 13.
- The color graphics adapter accepts only values from 0 to 7.
- EGA and VGA cards accept values from 0 to 7. The EGA/VGA BIOS automatically adapts the number selected to the size of the character matrix currently in use.

The functions we've discussed set the various parameters that control the mouse driver. The mouse driver also supports a group of functions that read the mouse's position as well as the status of the mouse buttons. These functions can be divided into two categories for reading external devices, such as the mouse, keyboard, printer, or disk drives. These categories are the polling method and the interrupt method. The mouse driver supports both of these methods.

Polling method

The polling method constantly reads a device within a loop. This loop terminates only when the desired event occurs. Since the execution of this loop requires the full capabilities of the CPU, there is usually not enough time left to perform other tasks.

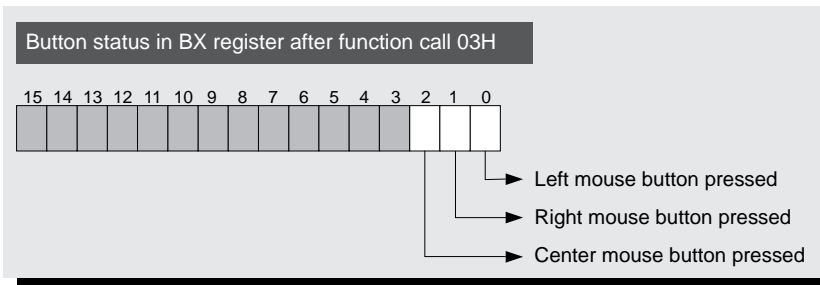
Interrupt method

The interrupt method has an advantage over the polling method because it allows the CPU to execute other tasks until the desired event occurs. Once this happens, the mouse driver calls an interrupt routine that reacts to the event and executes further instructions.

Function 03H:	Get cursor position/button status
---------------	-----------------------------------

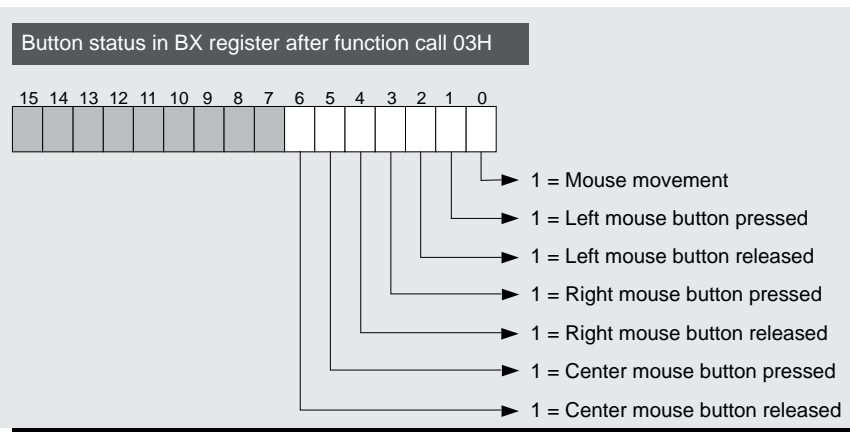
The polling method provides four functions that operate in conjunction with the mouse interface. These functions can be accessed through function 03H, which returns the current cursor position and mouse button status. Function 03H passes the horizontal cursor position to the CX register and the vertical cursor position to the DX register. Since these coordinates also refer to the virtual mouse screen, they must be converted to the text screen's coordinate system by dividing the components by eight, or by shifting the bits three binary places to the right.

The following illustration shows how the mouse button status is returned to the BX register. Only the three lowest bits represent the status of one of the two or three mouse buttons. The bit for the corresponding mouse button contains the value 1 when the user presses that mouse button during the function call.



Function 0CH:	Set event handler
---------------	-------------------

Function 0CH sets the address of a mouse event handler (interrupt routine). The function number must be passed to the AX register. The segment and offset address of the event handler must be passed to the ES:DX register pair. The event mask must be passed to the CX register. The individual bits of this flag determine the conditions under which the event handler should be called. The following illustration shows the CX register coding:



After executing the function, the mouse driver calls the event handler when at least one of the specified events occurs. The call is made using the FAR call instead of the INT instruction. It's important to remember this difference when developing an event handler; the handler must be ended with a FAR RET instruction instead of an IRET instruction. Similar to an interrupt routine, none of the various processor registers can be changed when they're returned to the caller. Therefore, the registers must be stored on the stack immediately after the call and the register contents must be restored at the end of the routine.

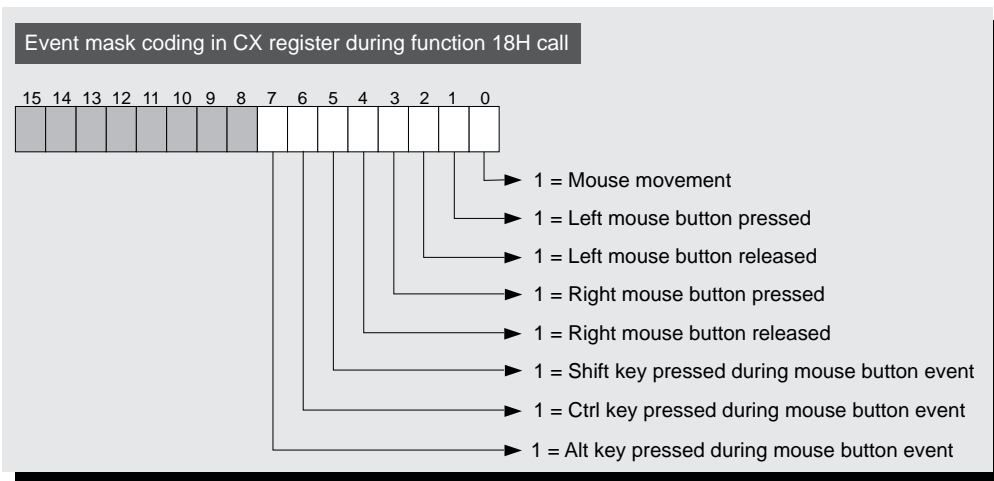
Information is passed to the event handler from the mouse driver through individual processor registers. The information about the event can be found in the AX register, in which each bit has the same significance as in the event mask during the call of function 0CH (see the previous table). Individual bits, which have no meaning for the event handler, may be set. For example, if the event handler should be called only when the left mouse button is activated (bit 1), bits 0 and 4 may also be set during the event handler call. This is possible because the mouse was moved and the right mouse button was released simultaneously.

The event handler can obtain the current button status from the contents of the CX register. The coding is identical during the call to the function 03H. Bits 0 to 2 represent the different mouse buttons. The current cursor position can be found in the CX and DX registers, which represent the horizontal and vertical positions. The position can only be set after conversion to the text screen's coordinate system.

During the development of an event handler, the DS register should point to the data segment of the mouse driver during the handler call, instead of the interrupted program. If the event handler accesses its own data segment, it must first load its address into the DS register.

Function 18H:	Install alternate event handler
---------------	---------------------------------

Function 18H allows you to install an event handler, which reacts to limited-range keyboard events as well as mouse events. This function signals an event if the **Ctrl**, **Alt** or **Shift** keys are pressed when a mouse button is pressed or released. This function is almost identical in register assignments to function 0CH. The event mask in the CX register has been extended by the three events, as shown in the following illustration:



Even during the call of such an alternative event handler, little changes in comparison with the event handlers that were installed by calling function 0CH. Only the content of the AX register must be interpreted differently, since its construction corresponds to the event mask previously shown.

Up to three alternative event handlers can be installed by calling function 18H. During the function 0CH call, the event handler indicated replaces the previously installed handler. Three different event handlers can be installed by calling function 18H three times. This is only valid if the three event handlers are equipped with different event masks. If an event mask passes to function 18H, which is already equipped with a handler, the new handler replaces the existing handler.

Sample Programs

This section contains programs in C and Turbo Pascal that demonstrate mouse access functions. These programs show the techniques for developing and installing an event handler, which is the most complicated part of mouse reading. Both programs include functions or procedures that call various mouse functions. These routines require little programming. They load the processor registers with the necessary values, then call interrupt 33H. Since the event handler needs the most programming, we'll discuss this in detail.

Installing an event handler in a higher level language program is difficult because it must meet certain requirements, which usually cannot be controlled by a programmer. The requirements are as follows:

- The event handler must be a FAR procedure, and must be terminated with a FAR RET instruction.
- The event handler must store the various processor registers during the call and restore them before completion.
- The event handler must load the segment address of the higher level language data segment into the DS register to provide access to global variables of the program.

Although these requirements can be met in some versions of Turbo Pascal, Turbo C, and Microsoft C, very complex programming is required. The traditional solution (write a routine in assembly language) is easier and faster to implement. Therefore, we wrote the event handler itself in assembly language, assembled the program, and linked the resulting object module to the higher level language program.

This assembler routine is named AssmHand. It stores the various processor registers on the stack after the call, then calls a C function or Pascal procedure named MouEventHandler. The AssmHand routine passes arguments provided by the mouse driver to the MouEventHandler routine. These arguments include:

- The event flag, which describes the event that caused the handler call.
- The current mouse button status.
- The current position of the mouse cursor.

This information is converted from virtual graphic screen coordinates into text screen coordinates (25 lines x 80 columns). The stack handles parameter passing. The C version of `AssmHand` must pass the arguments onto the stack in the reverse order of their declaration. After loading the DS register and calling the higher level language routine, these arguments must be taken from the stack again by incrementing the stack cursor by the memory requirements of the arguments (8 bytes). This is only required for the C version of the routine. The Turbo Pascal version performs this task on its own.

After calling this routine, the `AssmHand` routine returns the processor registers to the stack and passes control to the caller using a FAR RET instruction. The `AssmHand` instructions execute very quickly, but the handler itself may require more execution time than expected. This introduces the problem of recursion, since an event in connection with the mouse may recur during the handler execution. The `AssmHand` driver then must be recalled before the previous call is terminated.

To avoid this situation and the complications that can occur, `AssmHand` maintains a variable named `active` in its code segment. During execution this variable contains the value 1. Before setting this variable, the program tests if `active` already contains the value 1. This indicates the last call wasn't completed yet. If this situation occurs, the handler execution terminates immediately, thus avoiding recursion.

Even if this method avoids recursion problems, remember that it can produce its own problems. The suppression of the higher level language handler doesn't notice the event, because the handler wasn't called by the mouse driver. Although we offer the recursion trap as an option, we recommend that you program the higher level language handler as efficiently as possible to avoid using processor time. This will keep call suppression to a minimum.

The `AssmHand` handler

`AssmHand` must first be installed through function 0CH, using the `MouSetEventHandler` procedure/function. `MouSetEventHandler` is called by the `MouInit` procedure/function, which initializes the mouse module. This should be called by any application program as the first procedure/function of this module. The number of lines and columns of the display screen must be passed to it as arguments, to determine the size of an internal buffer needed for the various procedure/functions within the module.

This buffer can divide the screen into individual mouse ranges, each equipped with its own code, cursor mask, and screen mask. These mouse ranges are very important in mouse access. They permit the definition of objects such as sliders, command buttons, or menu items. As soon as the user moves the cursor to an object and presses a mouse button, the object executes a particular step in the program.

`MouDefRange` defines these ranges. The registration of these ranges occurs through the procedure/function `MouDefRange`, which must receive a cursor to a vector or array, and the number of elements stored there. These elements of the type `RANGE` describe a screen area and the cursor or screen mask assigned to the cursor as soon as it reaches this area. An area can comprise a single character or the entire screen. The user can define the array with individual area descriptors. The area code depends on the position of the descriptor within the array, and is provided automatically by the procedure/function `MouDefRange`. The first area has the value 0, the second the value 1, etc. The screen areas not covered by an area descriptor are assigned the code `NO_RANGE`.

During the creation of this array, especially during the definition of the cursor and screen mask in the `PtrMask` array, the C implementation provides helpful macros and constants. The Pascal program has functions and constants available for this purpose. The creation of a variable of the type `PTRVIEW`, stored in the `PtrMask` field within an area descriptor, is handled by the macro or function `MouPtrMask`. The cursor and screen mask for the character must be passed to `MouPtrMask` to define the cursor's appearance on the screen.

If `PtrSameChar` is indicated, the cursor appears as the character that it covers. If another cursor is desired, the cursor can be defined with `PtrDifChar`. When the call occurs, enter the ASCII code of the desired character for `PtrDifChar`. As a second

parameter, `MouPtrMask` receives the cursor's color from the cursor mask and screen mask. Many options for color are possible:

- `PtrSameCol` ensures the cursor assumes the color of the character currently overlapped by the cursor.
- `PtrSameColB` creates a cursor that assumes the color of the character currently overlapped. However, bit 7 of the attribute byte is set to 1 so the character either blinks or appears with a high-intensity background color.
- `PtrInvCol` makes the cursor appear in the inverse color of the character currently overlapped by the cursor.
- `PtrDifCol` displays the cursor on the screen in the color indicated by the code following `PtrDifCol`.

In addition to the different mouse areas specified through `MouDefRange`, a cursor can be assigned to the remaining screen, which is the area carrying the code `NO_RANGE`. A program can use `MouSetDefaultPtr` to obtain the cursor and screen mask of the cursor as a parameter of type `PTRVIEW`. The constants and macros or functions previously described can be used to create this parameter.

Changing the mouse cursor

The `MouEventHandler` changes the cursor and screen mask for each area. Since it's called for every mouse event (including mouse movement), it can determine the mouse area where the cursor is currently located. To make this happen as fast as possible, it tests if the mouse area contains the position of the cursor.

`MouEventHandler` uses the internal region buffer that was created by `MouInit` during the call. It reflects exactly the video RAM structure, and contains one byte for every screen position. Each byte contains the code of the area to which the screen position was assigned. The event handler can use the current position of the cursor as an index to this area buffer. A single memory access is enough to determine the mouse area in which the cursor is located. The area code found is stored in the global variable `MouRng`, and is used as an index to the array of the mouse descriptor from which it determines the cursor and screen mask for this area.

The higher level language event handler has another assignment that may be even more important. It controls the variable `MouEvent`, in which the current mouse events are stored. This task cannot be performed by simply copying the mouse events that were passed through `AssmHand` from the mouse driver. This only shows the current event, but no preceding events. If the user presses and holds the left mouse button, then presses the right mouse button, this results in two event handler calls. This signals each case of an active mouse button. The preceding call (the active left mouse button) is no longer recognized by the call, since it reports only the current event (the depressed right mouse button).

The event handler must isolate the various events that are reflected in the `EvFlags` variable, and accept only new events in the `MouEvent` variable. This variable reflects the current status of the mouse buttons, and the cursor's current movement or position. `MouEvent` can handle the most important mouse sensing tasks, waiting for the occurrence of a certain event (usually a pressed mouse button). `MouEventWait` waits for the occurrence of an event, which was specified by the bitmask that was passed earlier. This bitmask can be defined through the logical OR function with the constants shown in the table on the right.

The procedure/function can be instructed to wait for one or more of these events to occur. The AND or OR correspond to the logical comparisons of the same names. Which events occur can be sensed through the results of a bitmask in which the individual bits represent the various events, and through which the constants previously described can be sensed.

Constant	Task
<code>EV_MOUSE_MOVE</code>	Mouse movement
<code>EV_LEFT_PRESS</code>	Left mouse button pressed
<code>EV_LEFT_REL</code>	Left mouse button released
<code>EV_RIGHT_PRESS</code>	Right mouse button pressed
<code>EV_RIGHT_REL</code>	Right mouse button released

You'll find the following program(s) on the companion CD-ROM



MOUSEP.PAS (Pascal listing)
 MOUSEPA.ASM (Assembler listing)
 MOUSEC.C (C listing)
 MOUSECA.ASM (Assembler listing)

Mouse - PC Communication

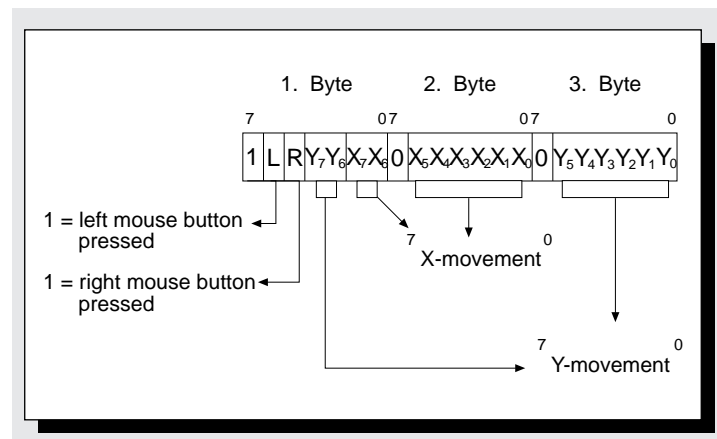
Let's briefly look at how the physical movements and button accesses of the mouse are translated into information the computer understands. The method of data transfer depends on the manufacturer of the mouse, and the way the mouse interfaces with the computer. A serial mouse operates differently from a bus mouse. Since the serial mouse is more common than the bus mouse, we'll discuss only the serial mouse.

The mouse driver determines whether a mouse is connected during its initialization. For a Microsoft mouse, the mouse driver sets the serial port's DTR (DTR = Data Transfer Ready) line to 1. The mouse recognizes this signal and sends the ASCII code for the letter M over the data line. The driver doesn't wait very long for this return message. This is why a Microsoft-compatible mouse may not always work with the Microsoft mouse driver. Although the mouse is set up to transfer data according to the Microsoft protocol, the driver may not recognize the mouse if the "M" isn't sent promptly.

Once successful handshaking has been established, the mouse uses the interrupt method to transfer its information to the driver. The mouse executes a hardware interrupt at the serial port as soon as it needs to inform the driver about a movement or button status change. The interrupts used are 0CH (COM1) or 0DH (COM2). The mouse driver diverts the interrupts to a routine of its own so it can read mouse data directly from the serial port. The data transfer for a Microsoft mouse uses the following parameters: 1200 baud, 7 data bits, no parity bit. Each mouse message consists of three bytes that indicate the mouse button status and the relative movement of the mouse since the last data transmission. The distance for mouse movements is measured in mickeys. Depending on the video card and mouse resolution, one mickey can be either be 1/200 or 1/400 of an inch.

As the following illustration shows, only 7 bits are available to describe mouse movement along the X and Y axes. This allows us to cover distances between -128 and +127 mickeys. At the very latest, the new mouse position must be sent when the mouse moves outside this range. Generally, a new message will be sent much sooner than this.

*Data transfer
format-Microsoft
serial mouse*



As the illustration above shows, the Microsoft format is limited to two mouse buttons. The highest bit is reserved for synchronization with the mouse driver. This prevents the mouse driver from getting the second or third byte of the message first when the hardware interrupt fails and the first one or two bytes are lost. The high bit in the first byte is always set to 1 and the high bits of the other two bytes are always set to 0.

Other mouse manufacturers that use three buttons use a different communication format (e.g., eight data bits instead of seven). This extra bit allows transmission of the third mouse button's status. The other principles of communication are basically the same; the mouse is still connected to a serial port and the serial port interrupts communicate with the driver.



Memory Expansion

When the IBM PC was being developed in 1980, its capabilities were quite advanced at the time. This was also true of its main memory size. The maximum size of 640K seemed so large in the early 1980s that no one could imagine what a user would do with so much memory. Thus, the first PCs were equipped with 64K, then 128K, and later 256K of memory. Today memory requirements continue to increase. The minimum amount of RAM for PCs has grown to 640K.

As we enter the age of Pentium microprocessors, graphical user interfaces, and multitasking operating systems, 640K simply isn't enough memory to use the full capabilities of your PC. But we've reached a boundary that cannot be crossed by simply adding more memory chips to the computer. This boundary is the 1 megabyte limit, set by the 8086 microprocessor's addressing capabilities.

Another important factor affecting memory is compatibility. To maintain program compatibility in machines ranging from the simplest XT to a fully equipped Pentium, the processors must be compatible and the memory layouts of each machine must meet certain standards. In this chapter we'll examine ways to access alternate forms of memory.

The illustration on the top of the following page shows the basic memory configuration of the PC. Notice only the first 640K can be used for RAM storage. Any memory beyond that point is reserved for video RAM, hardware enhancements and the BIOS.

Memory can easily be enlarged beyond the one megabyte limit. Today most computers come equipped with 4 megabytes or more of RAM but this memory cannot be addressed under DOS. Because DOS operates in real mode, memory beyond the 1 megabyte limit cannot be accessed.

There are many solutions for this RAM crisis. Memory needs can be handled by expanded memory and extended memory. These types of memory can increase memory capacity by many megabytes (providing the software can use this capacity).

Before we continue, let's clear up the confusion between expanded memory and extended memory. Since the two terms sound very similar, it's easy to forget which memory performs which task.

Expanded memory

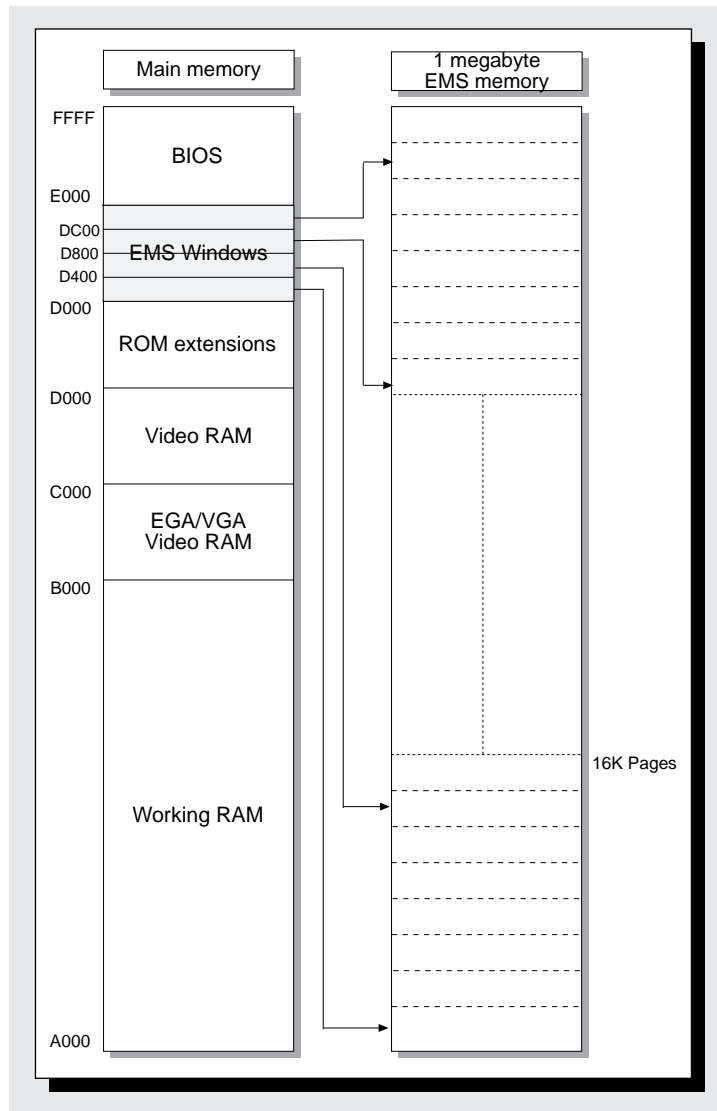
Expanded memory is the additional memory you'll find in PC/XT computers. Because of the 8088 processor, these computers are limited to 640K of working RAM. With expanded memory, RAM beyond the 640K boundary can be used. Remember that expanded memory expands RAM beyond 640K.

Extended memory

Extended memory is the additional memory found on 80286 and higher end computers. This memory extends beyond the 1 megabyte boundary.

Let's take a closer look at each memory type.

Accessing EMS memory of the LIM standard with the help of "windows"



Expanded Memory

A PC or a PC/XT machine is limited to 640K of conventional memory. 80286 based computers are limited to 16 megabytes of RAM. However, those 16 megabytes are only available when the PC is running in Protected mode (see Chapter 36 for more information), which makes the memory inaccessible to DOS programs.

Several years ago some leading PC firms worked together to devise a way to add more memory, which could also be accessed under DOS, to PCs, XTs, and ATs. These companies were Lotus (the developers of Lotus 1-2-3), Intel (manufacturer of PC processors), and Microsoft (developers of MS-DOS and OS/2). They developed the LIM standard; "LIM" represents the first letters of the company names.

This standard allows up to 8 megabytes to be added to a PC on an expansion card. Only 64K of this 8 megabytes is visible in the 1 megabyte address range of the 8088 processor, in a window called the page frame. Memory installed in this manner

is called expanded memory. This memory shouldn't be confused with the extended memory that goes beyond 1 megabyte on an AT. The entire system is referred to as the Expanded Memory System (EMS).

The LIM standard used the reliable trick of bank switching for memory access. Bank switching creates a small memory window, through which a portion of expanded memory can be accessed beyond the address space of the computer. The total addressable memory capacity may extend over several megabytes. The software works with the hardware, moving this window to permit access to the portion of memory currently needed. The remaining memory stays invisible to the program.

Opening the memory window

At least 64K in the 1 megabyte address space of the PC is unused by conventional memory, BIOS, video RAM, or other system expansions. The EMS developers decided to use this as a window into expanded memory. Usually this window lies at segment address D000H, but EMS hardware allows this window to be relocated.

Since this window is under the 1 megabyte memory limit, it can be accessed with normal assembly language instructions, similar to video RAM access. Both read and write accesses are possible. We'll look at specific examples of these accesses later in this chapter.

Page frame division

The page frame is further divided into 16K pages. This allows the programmer to access four completely different pages in EMS memory.

The EMS card's registers allow the programmer to set which pages of the EMS memory will be visible in the page frame. The address lines on the EMS card are programmed so the EMS pages are mapped into the page frame and appear in the 8088's address space. This process is the bank switching we mentioned earlier in this chapter.

Besides the hardware, the EMS also includes a software interface, which handles EMS register programming and other memory management tasks. This software interface, called the EMM (Expanded Memory Manager), provides a standard interface that you can use to access EMS cards from different manufacturers. This also applies to the extended EMS standard (EEMS) developed by AST Research, Quadram and Ashton-Tate, which surpasses the LIM standard.

The best known examples of EEMS are 386-To-The-Max (Qualitas), QEMM-386 (Quarterdeck), and Microsoft Windows. Windows' 386 enhanced mode makes its applications available in expanded memory, where Windows stores extended memory.

The above products are based on a special 80386 operating mode, called the virtual 8086 mode. This mode lets you move memory from above the 1 megabyte barrier into conventional memory, and give this memory the look and feel of a normal page frame. Compared to a conventional hardware implementation, this immediately provides several advantages:

- Without EMS emulation, this memory is accessible as extended memory.
- Because hardware addressing or I/O addressing isn't required, EMS access and switching between pages occurs very quickly.
- Lower cost (EMS cards usually cost more than software emulation).
- One more expansion slot is free than there would be with true EMS memory.

This EMS emulation is possible with the 80386 microprocessor and its successors; it's also possible on 80286 machines but speed is decreased. Many Asian AT manufacturers offer the option of using extended memory as expanded memory by using the NEAT (New Enhanced AT) chips available from Chips & Technologies Corporation. These chips allow hardware to view extended memory as expanded memory, without installing a special enhancement card.

The software interface between the EMM and a program resembles many other software interfaces found in the PC environment.

History of the LIM standard

While most programs appear on the market with a starting version number of 1.0, EMS Version 1.0 never existed; actually it never left the laboratory. When Intel and Lotus presented their proposal for an EMS standard in the spring of 1985, Version 3.0 already existed. Microsoft joined the other two firms shortly after this release because they could use a capability of this type for their own products.

Version 3.2 was released in the fall of 1985. The EMS standard became the LIM standard. A few weeks later, a group of firms comprising AST, Quadram, and Ashton-Tate presented their own EMS standard, called the Enhanced Expanded Memory Specification (EEMS). This standard was based on EMS Version 3.2, but offered few advantages over EMS other than the ability to use page frames larger than 64K. EMS 4.0 now supports such capabilities, and EEMS is part of computing's past.

EMS 3.2 became a big success and received support from many well-known manufacturers. Lotus promoted the sale of EMS memory cards because EMS enabled Lotus 1-2-3 to handle very large spreadsheets. TSR (Terminate and Stay Resident) programs, RAM disks, and other utilities also used expanded memory. Microsoft eventually provided EMS support beginning with MS-DOS Version 4.0.

EMS Version 4.0

With its 14 functions, EMS Version 3.2 satisfied all the needs of software developers. EMS Version 4.0, released in the fall of 1987, had 58 functions available for enhanced hardware support. This release included the following features:

- Memory support for 32 megabytes instead of 8 megabytes.
- Ability to generate EMS windows at any location in addressable memory.
- Any size EMS windows, instead of the 64K required by earlier versions.
- EMS pages that can be protected from a system crash or an accidental reset, which increased the value of EMS based RAM disks.

Version 4.0 makes greater demands on the EMS hardware than earlier versions. This may be why Version 4.0 hasn't become very popular and Version 3.2 remains the standard for EMS programming. Therefore, we'll concentrate primarily on EMS Version 3.2.

EMS Version 3.2

The Expanded Memory Manager (EMM) enables the programmer to access EMS memory. We mentioned the major changes between Versions 3.2 and 4.0 earlier in this chapter. However, you'll probably use only a few Version 3.2 functions in EMS programming.

We'll concentrate on EMS Version 3.2 in this book because this version is considered the standard for expanded memory. If you want more information about Version 4.0, EMS Version 4.0's functions are documented in the Appendices on the companion CD-ROM.

The EMM

Similar to DOS interrupt 21H, which provides a standard interface to the operating system functions, EMM functions can be called through interrupt 67H. Before a program tries to use EMS memory and the corresponding EMM, it should first check to ensure that an EMS is installed. If it doesn't do this and there is no EMM, the results of a call to interrupt 67H are completely unpredictable. The call simply may not work or the system may crash.

To prevent this from happening, a program that uses the EMS must first check to ensure that EMS exists. This can easily be done when you consider the EMM is bound into the system as a normal device driver when the computer is booted. So, it naturally has a driver header that precedes it in memory and defines its structure for DOS. The name of the driver is found at address 10 in the driver header. The LIM standard requires that this name be EMMXXXX0.

The example programs at the end of this chapter test for this name by first determining the segment address of the interrupt handler for interrupt 67H. If the EMM is installed, the segment address points to the segment into which the EMMXXXX0 device driver was loaded. Since the driver header is at offset address 0, relative to the start of this segment, we simply compare the memory locations starting at 10 with the name EMMXXXX0 to determine whether the EMS memory and the corresponding EMM are installed.

Once this is verified, this memory can be accessed in three steps:

1. As conventional memory must be allocated with a DOS function, a program must first allocate a certain number of EMS pages for itself from the EMM. The number of pages to be allocated depends on both the memory requirements of the program and how much EMS memory is available.
2. If the desired number of pages were successfully allocated, the specified pages must first be loaded into one of the four pages of the page frame so data can be written into them or read from them. This results in a mapping between one of the allocated pages and one of the four physical pages within the page frame.
3. When the program ends or it's finished using EMS, the allocated pages should be released again. If this isn't done, the allocated pages will still be owned by the program (even after it ends) and cannot be given to other programs.

As with DOS interrupt 21H, the function number of an EMM call must be loaded into the AH register before the interrupt call. After the function call this register contains the error status of the function. The value 0 signals the function was executed successfully, while values greater than or equal to 80H indicate an error.

About errors

The Appendices on the companion CD-ROM mentions the error codes which are listed in the error descriptions. However, you should be aware of one particular error. If the value 84H is in the AH register after a call to EMM interrupt 67H, this indicates that an invalid function number was passed in the AH register.

The table on the right lists the functions which are needed for a transient program to access the EMS memory.

To ensure the EMS hardware and the EMM operate properly, check the EMM status before allocating EMS memory. This is done with function 40H, which only requires the function number in the AH register. If it returns the value 0 in the AH register, then everything is OK and you can start working with the EMS memory.

Function	Task
40H	Get EMM status
41H	Get segment address of page frame
42H	Get number of pages
43H	Allocate EMS pages
44H	Set mapping
45H	Release EMS pages

Limits to EMS allocation

The number of allocatable EMS pages is limited by the number of free pages. So, you should ensure the memory requirements of the program don't exceed the available memory. Here we can use function 42H, which returns the number of free EMS pages. This function requires only the function number and returns the number of unallocated pages in the BX register. It also returns the total number of installed EMS pages in the DX register.

If enough EMS memory exists for our program, or if the memory requirements are adapted to the available memory, then we can allocate the memory. The number of pages to be allocated must be passed to function 43H in the BX register. If the requested number of pages is successfully allocated (AH register contains 0 after the function call), the caller will find a handle to the allocated pages in the BX register. This handle, which must be used to access the allocated pages, identifies the caller to the EMM. The caller must save this handle. If the handle is lost, the allocated pages cannot be accessed and can no longer be released. A program can call this function several times to allocate multiple logical page blocks.

Once we have the page handle we can start accessing the pages. The handle is passed to the appropriate functions in the DX register. This also applies to function 44H, which maps a logical page to one of the four physical pages of the page frame.

The number of the logical page is passed in the BX register and the physical page number in the AL register. Note that both specifications start at zero. So, if you've allocated 15 pages, then the numbers of the logical pages run from zero to 14.

Once the appropriate page is in the page frame, it can be accessed like normal memory. The offset address of the start-of-page is calculated from the physical page number but the corresponding segment address must be determined with an EMM function. Since this address doesn't change while working with the EMS memory, you can read it once at the beginning of the program and then save it in a variable. Function 41H returns the segment address of the page frame in the BX register. When you're finished using the EMS, you must return the allocated pages to the EMM. Simply pass the page handle to function 45H.

Besides these six functions, which a normal program can use to access EMS memory, the table on the right shows more functions which can be useful in certain circumstances.

Function	Task
46H	Get EMM version number
47H	Save current mapping
48H	Reset saved mapping
49H	Get number of EMM handles
4AH	Get the number of pages allocated to a handle
4BH	Get all handles and numbers of allocated pages

Version numbers

Reading the EMM version number is important because the LIM standard has changed slightly since it was introduced. Some functions are no longer supported and new functions have been added. The functions presented here are from Version 3.2, which has been replaced by Version 4.0. Version 3.2 represents a good compromise not only because it's widely used, but also because it's completely compatible with Version 4.0. If you don't want to support earlier or later EMS versions in your program, you should check the version number at the start of the program. The version number will be returned in the AL register after a call to function 46H. It's encoded as a BCD number.

Functions 47H and 48H are important for TSR programs that want to use the EMS memory. When a TSR program interrupts a transient program and places itself in the foreground, it must consider the interrupted program may have been using EMS memory and had created a certain mapping. Since this mapping shouldn't be changed when returning to the interrupted program, it must be saved when the TSR is activated and then restored when the TSR exits. Function 08H saves the current EMM mapping and function 09H resets the saved status. The handle of the function must be passed to both functions. In this case, it's the handle of the TSR program instead of the handle of the interrupted program.

Since the last three functions are important only to the memory manager, we won't discuss them here. See the Appendices for more information.



You'll find the Appendices on the companion CD-ROM

Sample programs

The EMMC.C and EMMP.PAS programs on the companion CD-ROM show how to use EMS memory. There isn't an assembly language program because, theoretically, calls to the EMM functions simply involve loading variables and constants into registers and calling the EMM interrupt 67H. By using the information in the Appendices (see the companion CD-ROM), you should be able to write an assembly language program that uses the EMS. There isn't a BASIC program because EMS memory is intended to be used with complex and memory-intensive applications for which BASIC isn't suited.

Since the two programs are almost identical, we'll discuss only the basic program structure. The programs provide several functions and procedures that can be used to access the various EMM functions. Both programs also contain a function called EMS_INST (or EmsInst), which determines whether an EMM is installed.

In Pascal we encounter a problem because a pointer must be loaded with an address consisting of separate segment and offset addresses. Since this isn't possible in Pascal, there is an INLINE procedure, called MK_FP, that (like the C macro of the same name) combines a segment and an offset address into a (FAR) pointer. Since this is a FAR pointer, the page frame isn't in the program's data segment. So, it cannot be addressed by the DS register. This isn't a problem in Turbo Pascal because the code is generated to work with FAR data pointers. In C, we must ensure the program is compiled in a memory model that uses FAR pointers for data. This occurs in compact, large, and huge models.

The main program firsts tests to determine whether EMM is present. Then it uses various functions to obtain status information about EMS memory, which it displays on the screen. Next, a page is allocated and mapped to the first page (page 0) of the page frame. The current contents of video RAM are copied into this page and the video RAM is then erased.

After the copy procedure, a message is displayed for the user and the program waits for a key to be pressed. Then it copies the old screen contents back to video RAM from page 0 of the page frame and the program ends.

This program shows the contents of a page in the page frame can be treated like ordinary data. After you've created a pointer to the corresponding page, you can manipulate the data on this page, including complex objects like structures and arrays, like any other data. It's important to ensure that your objects fit on one page or that you don't forget to change pages or load a new page into the page frame to access larger objects.

You'll find the following program(s) on the companion CD-ROM



EMMC.C (C listing)
EMMP.PAS (Pascal listing)

Extended Memory

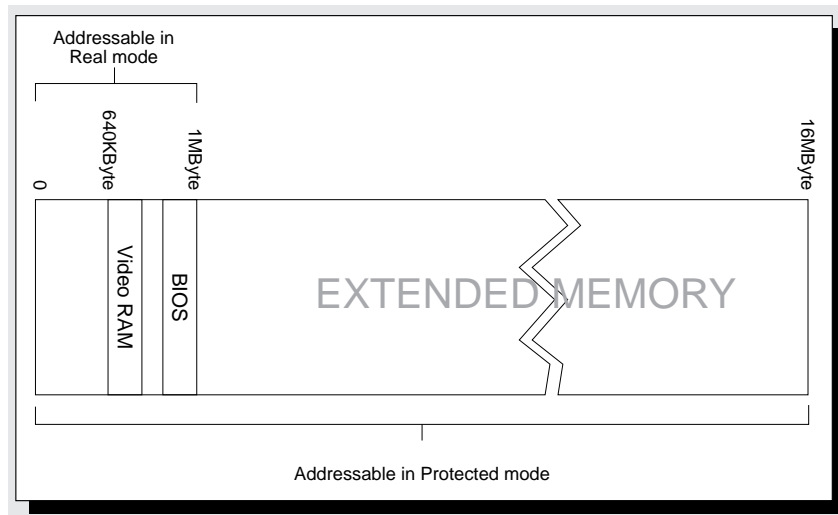
Although EMS access must be performed in portions, placing the PC in protected mode allows access to all of extended memory. In real mode, the extended memory isn't used because the processor cannot address memory beyond the 1 megabyte limit. This explains why 8088 and 8086 based systems cannot use extended memory. These microprocessors recognize only real mode and cannot switch to protected mode.

Changing a single bit in the processor's flag register enables protected mode. Once this is done, a program can access up to 16 megabytes of extended memory, but crashes immediately. This occurs because of the mechanism through which the 80286 and its successors address memory in protected mode. This is completely different from memory access in real mode.

When in protected mode, the processor operates through segment descriptors instead of segment addresses. These segment descriptors point to local and global segment descriptor lists. However, DOS cannot handle this access because it's limited to real mode.

The proper descriptor lists must be created and initialized before you can switch to protected mode. To do this, you must be familiar with assembly language programming and understand how the processor operates in protected mode.

Extended memory address space of a PC



Some BIOS functions and the XMS drivers provide help for extended memory access. The XMS drivers are able to handle extended memory better than the BIOS functions. The drivers can ensure that extended memory is shared instead of limiting all extended memory to a single program.

The fight for access rights to extended memory is a significant problem, which we'll discuss later in this chapter. This problem is especially noticeable if you access extended memory directly. The lowest 64K of extended memory is accessible from real mode (more on this later).

In the following sections, we'll discuss access using BIOS functions, the High Memory Area (HMA), and XMS functions.

Extended memory access from BIOS

Extended memory can be accessed only if it exists. Normally this applies to all computers that have 1 megabyte of RAM because usually only 640K or 512K in the area below the 1 megabyte limit are used. The rest is located immediately above this limit and is therefore available as extended memory.

Function 88H of BIOS interrupt 15H returns the extended memory size. Interrupt 15H was originally intended for the cassette recorder interface (the cassette recorder was the original mass storage device used on the PC). When disk drives replaced the cassette recorder for mass storage, the cassette recorder functions were no longer used.

Interrupt 15H is used for extended memory and joystick reading (see Chapter 10 for information on joysticks). Place function 88H in the AH register. The result, which is placed in the AX register, indicates the extended memory size in kilobytes.

Now that we know extended memory exists on the system, how can we access it? Function 87H moves blocks of memory within the total memory space. This means that blocks of memory can be moved from the area below the 1 megabyte limit to the area above the 1 megabyte limit and vice versa. However, the function shouldn't be used for the latter, since its call is complicated and has other disadvantages. To access memory beyond the 1 megabyte barrier, the processor must be switched to protected mode. Function 87H requires very comprehensive information, since the 80286 processor is more difficult to program in protected mode than in real mode (8086 emulation under DOS). At the end of this section you'll find a program that demonstrates how to use function 87H.

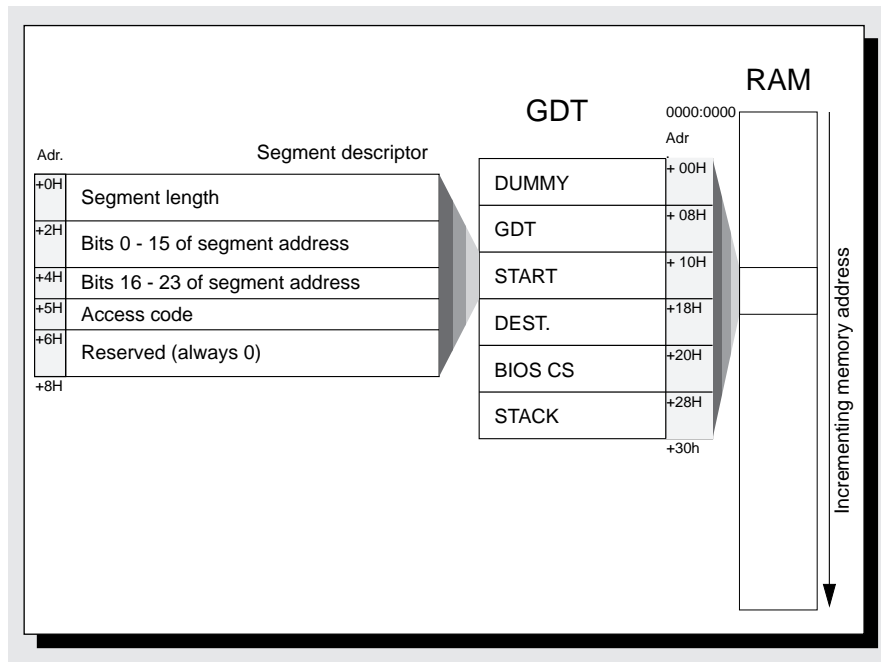
The function number 87H first must be passed to the AH register, then the number of the words to be moved (only words only, not bytes) must be passed to the CX register. A maximum value of 8000H corresponds to a maximum value of 64K.

Global Descriptor Table

The ES:SI register pair receive the address of the GDT (Global Descriptor Table) (Global Descriptor Table (GDT)), which must be installed in the user program. The GDT describes the individual memory segments of the 80x86 in protected mode. The segments in protected mode are exempt from the limitations made in real mode. While segments can only start at memory locations divisible by 16 in real mode, protected mode segments may start at any memory location. Also, protected mode segments may be any size from 1 byte to 64K.

Another protected mode innovation is the access code defined for every segment. This code indicates whether the segment described is a data segment or a code segment (only code segments can be executed). The access code also contains information on access priority and whether access is even permitted. Each segment descriptor consists of 8 bytes. During its call, function 87H expects that six segment descriptors have been prepared in the GDT (i.e., memory space reserved for them). The illustration below shows which segment descriptors are involved and the construction of a segment descriptor.

Segment descriptor structure as seen by function 87H



Since the BIOS functions fill out the other descriptors, we're only concerned with the segment descriptors designated as "start" and "destination". The start descriptor describes the segment, from which the data are taken. The destination descriptor describes the segment into which the data are copied.

The length of both segments can be 0FFFFH (64K decimal), even if fewer bytes (or words) copy over in the process. If a lower value is indicated, the number of bytes (number of words multiplied by 2) to be copied cannot exceed this amount. Otherwise, the processor notices an access across a segment boundary during copying, which triggers an error.

The address of the two memory areas must be converted to a (physical) 24-bit address. The lower 16 bits of this address enter the second field of the segment descriptor and the upper 8 bits enter the third field. Access code 92H can be used, which informs the processor the described segment is a data segment with the highest priority, the segment exists in memory, and the segment can be written. The last field of the descriptor maintains compatibility with the 80386 processor this field should always contain the value 0.

Although the address of the user program's buffer remains fixed, the address beyond the 1 megabyte boundary, to which data should be copied, can be freely selected (subject to RAM availability). The following table shows the addresses of the various 1K blocks beyond the 1 megabyte border as 24-bit addresses:

0K	=	100000H	124K	=	11F000H
1K	=	100400H	125K	=	11F400H
2K	=	100800H	126K	=	11F800H
3K	=	100C00H	127K	=	11FC00H
4K	=	101000H	128K	=	120000H
5K	=	101400H	129K	=	120400H
6K	=	100800H	130K	=	120800H
7K	=	100C00H	131K	=	120C00H
8K	=	102000H	132K	=	121000H
9K	=	102400H	133K	=	121400H

.....
.....

60K = 10F000H	252K = 13F000H
61K = 10F400H	253K = 13F400H
62K = 10F800H	254K = 13F800H
63K = 10FC00H	255K = 13FC00H
64K = 110000H	256K = 140000H
65K = 110400H	257K = 140400H
66K = 110800H	258K = 140800H
67K = 110C00H	259K = 140C00H
68K = 111000H	260K = 141000H
69K = 111400H	261K = 141400H

After the function call, the carry flag indicates the success of the function call. If the carry flag is set, an error occurred. The value in the AH register indicates the cause of the error (see table on the right).

A disadvantage of this function is that, while the processor is in protected mode, all interrupts must be suppressed. While in protected mode, BIOS interrupts (e.g., timer or keyboard) can be called, but these routines were designed to be used in real mode only. So, these interrupts may not work properly in protected mode.

Error number	Cause of error
AH = 0	No error (carry flag reset)
AH = 1	RAM parity error
AH = 2	GDT defective at function call
AH = 3	Protected mode could not be properly initialized

This disadvantage is evident when you call the timer. Since its interrupts are suppressed, protected mode doesn't keep track of the time. So time remains frozen for a moment. If programs frequently call function 87H, the clock may slow down by 20 or 30 seconds per day. However, since the clock can be reset to the proper time with software, most of these disadvantages can be avoided.

The execution speed of this function (slow) is more important than interrupt suppression. This is especially true for ATs, which are equipped with an 80286 processor. Although ATs can easily be switched to protected mode, switching back to real mode is more difficult. Since an instruction that smoothly switches the system back to real mode doesn't exist, a processor reset must be used.

An 80286 reset can only be triggered through the keyboard controller. A six millisecond delay occurs between the time the controller receives the reset instruction and the time the controller responds to the instruction. Those six milliseconds can be an eternity in a 10 MHz, 12 MHz, or 16 MHz computer. Unfortunately, this still isn't the solution. Although the processor does change to real mode after the reset, the system begins program execution with the boot code from BIOS.

To prevent this reboot process, a memory location in the BIOS RAM receives a code before triggering the reset. This code informs BIOS of the purpose of the reset. BIOS then returns to the caller of function 87H. However, this process requires a lot of computer time.

PS/2 systems handle this shift more smoothly, even if the systems are equipped with 80286 processors. The PS/2 includes special circuitry that makes a smooth transition from protected mode to real mode.

The 80386, 80486 and Pentium machines switch between modes easily, proving that Intel Corporation made the appropriate improvements to their processors.

Sample programs

The EXTP.PAS and EXTC.C programs in Pascal and C copy data between buffers in extended memory.

You'll find the following program(s) on the companion CD-ROM



EXTC.C (C listing)
EXTP.PAS (Pascal listing)

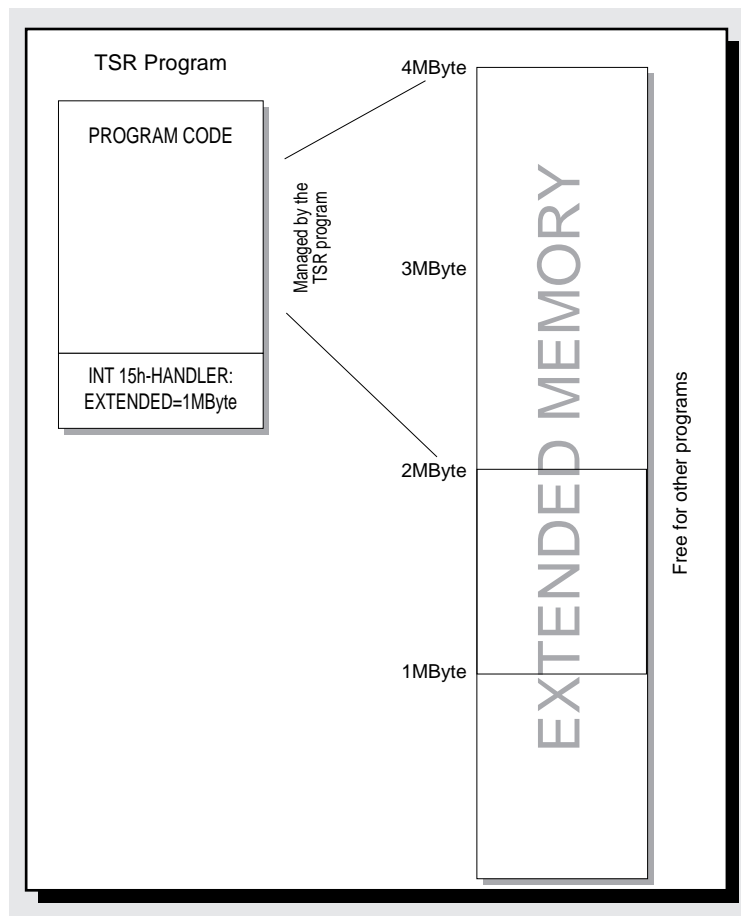
Conflicts in extended memory

Theoretically, extended memory should be shared by programs. However, some cache programs or other utilities may want to use all existing extended memory. This can cause memory overwriting by other programs and system crashes. This problem is caused by a lack of control. A program that can help other programs coexist in extended memory is needed. The BIOS causes this problem. BIOS function 88H informs every program that all of extended memory is available. The BIOS cannot allocate single memory blocks.

The XMS standard provides some solutions to this problem (more on this later). This standard, which was developed in 1988, is based on the XMS standard. Two different procedures help avoid collisions in extended memory, but don't always succeed. The first and most powerful method (the INT 15 method) redirects the interrupt vector of interrupt 15H so it points to its own handler instead of the original interrupt handler in ROM BIOS. The new interrupt 15H handler should concentrate on controlling function 88h, which checks the size of extended memory. The handler redirects the interrupt call to the original handler as soon as the function number confirms that another function should be called.

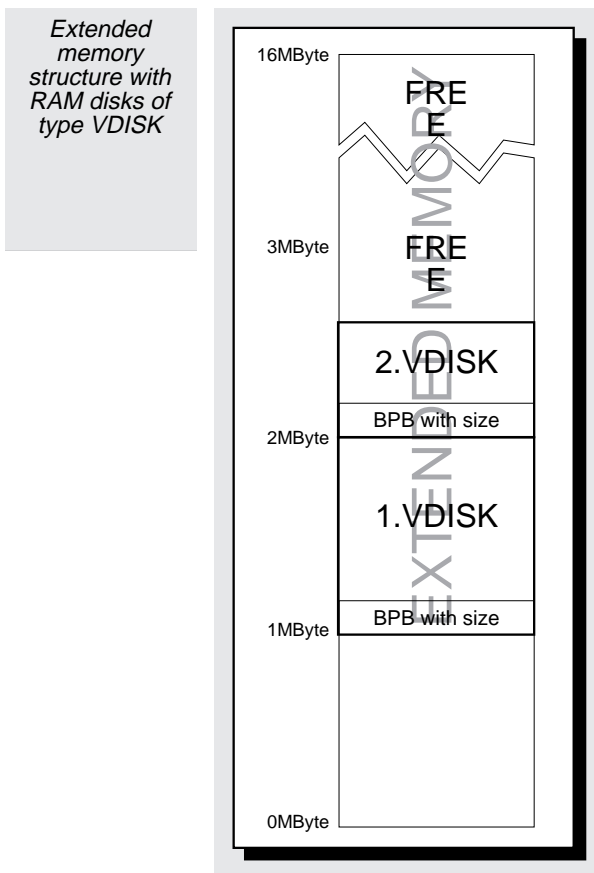
If function 88H is called, the caller receives the amount of extended memory currently available (not the current size of extended memory). Function 88H subtracts the amount of extended memory being used from the total extended memory size. This ensures the caller uses only the available extended memory. Because the caller must also assume the extended memory starts at the 1 megabyte border, the previously installed program must protect itself from the caller. The existing program is located at the end of extended memory instead of at the 1 megabyte limit. The caller views the starting location of the installed program as the end of extended memory.

*Extended
memory access
by redirecting
interrupt 15H*



The second method of extended memory control is frequently used by developers and tends to create new problems in programming. This VDISK method was first used in the VDISK device driver. VDISK is a RAM disk that can use extended memory for file storage. You'll find VDISK available on MS/PC-DOS starting with Version 3.0. VDISK stores its data starting at the 1 megabyte border, instead of isolating itself from other programs. VDISK won't overwrite memory reserved using the INT 15 method. However, programs called after VDISK will overwrite VDISK unless the subsequent programs can use the area at the end of extended memory. This is only possible through extensive testing, which are based on a knowledge of memory design under DOS.

The VDISK file header has a structure that's peculiar to all mass memory devices operating under DOS. This header contains status information and a small data structure, which can be designated as a BIOS Parameter Block (see Chapter 14 for more information about BPBs). This block lets you calculate the size of a storage medium and the length of the RAM disk currently in extended memory.

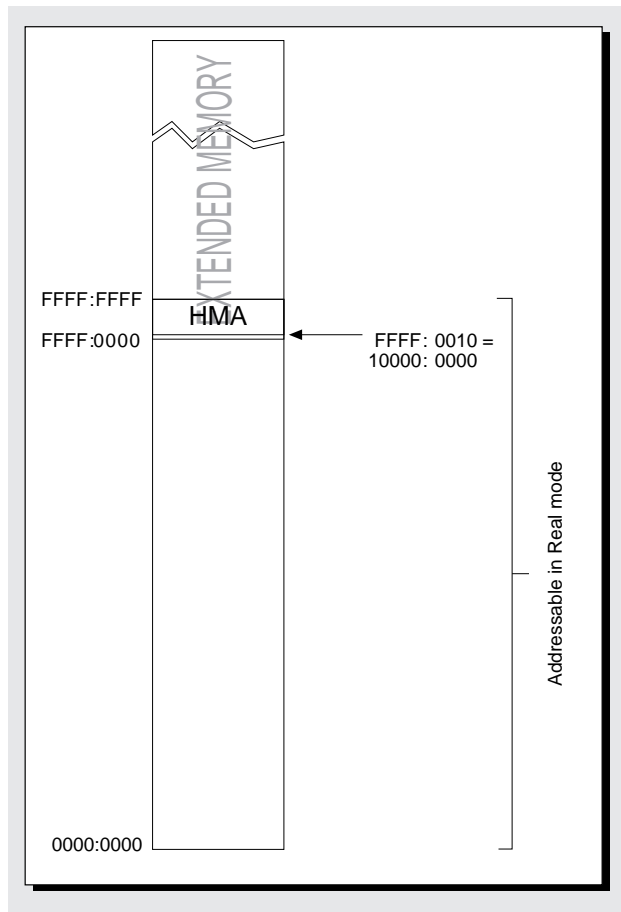


Multiple RAM disks complicate the search for available extended memory. We recommend that you always use an XMS driver for extended memory access, because this search doesn't work for every available type of RAM disk (more on this later).

Direct HMA access from real mode

The High Memory Area (HMA) is the first 64K of extended memory. The HMA is the only portion of extended memory that can be accessed (indirectly) from real mode without switching to protected mode. Although various sources credit either Microsoft or Intel with developing the HMA, the additional 64K for real mode is a great discovery.

Crossing the 1 megabyte barrier in real mode thru segment FFFFH



How can we access memory locations that actually lie outside the address space of the microprocessor? The answer lies in the way 80x86 processors form physical addresses, in real mode, from a segment address and offset address. Remember the segment address is multiplied by 16, then the segment and offset addresses are added. If the last segment in the address space of the PC (segment FFFFH) is the segment address, this places offset address 000FH in the physical addresses beyond the 1 megabyte limit. This places you in extended memory in real mode, and you can still access the conventional RAM below the limit.

Since the HMA starts with offset 00010H instead of offset 0000H, you'll find the HMA is smaller than 64K (65,520 bytes). This memory is more than enough to store data and short TSR programs.

Steps required for HMA access

A program that accesses the HMA should first ensure that HMA access is possible. If an 80x86 processor is located, we can continue because this indicates the A20 address line and extended memory exist. Next, calling BIOS function 88H of interrupt 15H determines how much extended memory exists. HMA access is possible only if a minimum of 64K is available in extended memory.

Address line must be disabled from interfacing with the keyboard controller. BIOS must enable address line A20 before HMA access can occur. This address line is usually disabled because some programs rely on address overflow at the 1 megabyte limit. This wraparound at address FFFF:0010 usually returns you to the start of conventional memory (address 0000:0000).

Address overflow with address line A20 disabled

Enabling address line A20 affects the keyboard controller as well as the switch from protected mode. On a system reset, the A20 switch occurs through the keyboard controller's output port. Bit 1 of this port must be set to 1 to make the A20 address line transparent, and set to 0 if a memory overflow occurs at the 1 megabyte limit. This bit cannot be easily accessed the keyboard must receive the instruction to disable access to the output port. Also, the controller requires specific parameters in communication. Refer to Chapter 14 for detailed information on how this communication works.

A normal PC with an ISA (Industry Standard Architecture) bus uses the keyboard output port for controlling the status of address line A20. With PS/2 systems and computers upgraded to an 80386 using the Intel InBoard, this line must be released using other methods.

A test should confirm the condition of the A20 address line. This is important because if this line isn't free, access to HMA cannot occur. With a small trick this can be determined, without a doubt, because a blocked line indicates an identical memory area through the address overflow at the 1 megabyte limit, such as the memory areas starting at FFFF:0010 and 0000:0000. By comparing the first bytes (e.g., the first 256) in this area, equality can be determined between the two, which proves that A20 is disabled.

Three routines are needed before you can work with the HMA:

- A routine to determine the existence of at least 64K of extended memory.
- A routine to switch address line A20.
- A routine to test address line A20 through memory comparison.

Demonstration programs

The HMAP.PAS and HMA.C demonstration programs on the companion CD-ROM below contain all three routines for HMA access. These programs, written in Pascal and C, provide a foundation on which you can build other HMA programs. Since all future HMA access should occur through the XMS driver (see Section 12.2.4), these demonstration programs don't test for VDISK compatible RAM disks. If this test doesn't exist, HMA access could overwrite the contents of the RAM disk. So, before starting one of these programs, ensure that a RAM disk doesn't exist in external memory. If a VDISK device or cache program exists, remove them, then run the demonstration program.

Both programs are based on the three routines mentioned above. We call these routines HMAAvail, Gate20, and IsA20On. These programs couldn't be developed in high level languages. So, we wrote them in assembly language and prepared them as INLINE commands (for the Pascal implementation) and as an assembly language module named HMA.C.ASM (for the C implementation). Both try to access HMA. If HMA cannot be accessed, the programs display an error message. A pointer of type FAR points to the beginning of the HMA. The Pascal implementation automatically creates FAR pointers, while the C implementation declares a FAR pointer.

If this pointer contains the address FFFF:0010, the HMA is accessible. Both programs perform a simple memory test, in which they fill the HMA with a constant value, then read this memory. Errors (if any) are then documented by the programs. It's unlikely errors will appear because any RAM failure would have been detected by the BIOS during bootup.

After this test, the program disables the A20 address line and suppresses HMA access. This must occur because other programs need the 1 megabyte limit for determining memory overflow. Remember the following rules for HMA access. First, avoid passing pointers that indicate DOS and compiler runtime library routines that refer to the HMA themselves. If you do this, these routines may normalize the pointer.

Normalization changes the segment of a pointer so the offset becomes less than 16, but the original memory location is still accessed as before. This results in an overflow of the segment address, which is then set to 0000H. You should avoid the normalization process.

Normalization of HMA pointers

	Segment address	Offset address
Original pointer	FFFF	010A
Normalization	+ 0010	- 0100
Normalized pointer	000F	000A

Also, diskette and hard drive operations shouldn't directly access HMA. The software often may cause problems in the DMA controller. This occurs because the DMA controller doesn't know how to properly handle addresses above the 1 megabyte limit. If you follow these rules, you shouldn't have any problems with direct HMA access. Remember these demonstration programs are simple examples. In the following pages we'll describe the XMS driver and how HMA is used in commercial programs.

You'll find the following program(s) on the companion CD-ROM

HMAC.C (C listing)
HMAP.PAS (Pascal listing)
HMACA.ASM (Assembler listing)

The XMS standard

The EMS standard contains a standard software interface, but extended memory existed for years without a software standard. This prompted many software manufacturers to develop their own versions. In 1988, Microsoft, Intel, Lotus and AST Research developed the Extended Memory Specification (XMS). The XMS standard defines a software interface that allows multiple programs to access extended memory and other memory areas simultaneously. The following types of memory accesses are supported:

- The HMA, which includes the first 64K of extended memory and extends from 1024K to 1088K.
- Four Extended Memory Blocks (EMB), which appear in extended memory starting at 1088K (thus avoiding conflict with the HMA).
- Upper Memory Blocks (UMB), which lie in conventional RAM between 640K and 1024K.

HIMEM.SYS from Microsoft Corporation is probably the most widely known XMS driver. This driver has been included with recent versions of MS-DOS and Windows. The system calls HIMEM.SYS during bootup, through the CONFIG.SYS file. Besides MS-DOS and Windows, you can find XMS drivers in some memory management programs designed for 80386, 80486 and Pentium computers. The XMS interface usually supports the LIM standard as well as extended memory.

Some examples of XMS systems are Qualitas' 386-To-The-Max, Quarterdeck's QEMM and Microsoft's EMM386.EXE, which is packaged starting with MS-DOS 5.0. These utilities can designate extended memory as Upper Memory Blocks in the range between 640K and 1024K (provided this range isn't allocated to video RAM, hardware enhancements, or ROM BIOS). The UMBs offer the developer RAM that can be addressed as if the UMBs are conventional RAM.

Few 80286 based systems without this driver can access Upper Memory Blocks, and not all XMS drivers will support 80286 systems. Exceptions to this rule use the NEAT chips from Chips & Technologies. A SETUP program lets the user configure memory, including UMBs. Before you can access XMS functions, you must determine whether an XMS driver is available. Interrupt 2FH is accessed by the XMS driver, similar to SHARE, APPEND, and PRINT in DOS. Call interrupt 2FH with function code 4300H in the AX register. If an XMS driver exists in the system, this function returns the value 80H to the AL register. Any other value indicates that an XMS driver isn't available and that XMS functions cannot be accessed.

XMS functions are called by a FAR CALL instruction, instead of an actual interrupt. Consequently, the XMS handler's address is needed for the call. This XMS handler is sometimes called the Extended Memory Manager (XMM). Interrupt 2FH

releases this address only if an XMS driver is actually installed, and if the first call of this interrupt returned the value 80H in the AL register.

Call interrupt 2FH and place the value 4310H in the AX register. The function returns the XMM's segment register to the ES register, and the XMM's offset register to the BX register. All XMS routines are accessible through this address. As the following table shows, 18 different XMS functions currently exist, of which the last three aren't available to all XMS drivers.

Function	Task	Function	Task
00H	Determine XMS version number	09H	Allocate Extended Memory Block (EMB)
01H	Allocate High Memory Area (HMA)	0AH	Free allocated Extended Memory Block (EMB)
02H	Free High Memory Area (HMA)	0BH	Move Extended Memory Block (EMB)
03H	Globally enable address line A20	0CH	Lock Extended Memory Block (EMB)
04H	Globally disable address line A20	0DH	Unlock Extended Memory Block (EMB)
05H	Locally enable address line A20	0EH	Get EMB handle information
06H	Locally disable address line A20	0FH	Resize Extended Memory Block (EMB)
07H	Query status of address line A20	10H	Allocate Upper Memory Block (UMB)
08H	Query free extended memory	11H	Free allocated Upper Memory Block (UMB)

Pass the XMS function number in the AH register to call the function. Other information may be placed in the other processor registers, but this differs among functions.

Almost all functions return a status code in the AX register. This status code provides information about the success of the operation. The value 0001H indicates successful execution, while 0000H indicates an error. If an error occurs, the BL register contains the error code, which provides details about the error. See the Appendices of this book for error codes and their meanings.

Next you should check the version number of the XMS driver by using function 00H. This function places the XMS version number in the AX register and the internal revision number in the BX register. The internal revision number indicates minor changes among drivers and is less important than the version number. Both items are returned as BCD numbers; the higher level byte accepts the version number preceding the decimal point and the lower byte accepts the internal revision number (the numbers following the decimal point). For example, 0200H represents Version number 2.0. XMS Version 2.0 is the lowest version you should own. If you have a Version 1 XMS, you should replace it.

Drivers with version numbers of 2.0 are most common, although some Version 3.0 XMS systems exist. We don't know the differences between the two versions, except that Version 3.0 was intended for 80386 memory management.

If you want to use the extended memory in your program as EMBs, use function 08H to determine the size of extended memory. This function returns the total amount of free extended memory (in kilobytes) in the DX register, and the size of the largest free EMB (in kilobytes) in the AX register. Use these values with caution; the HMA (64K in length) is included in calculating both values, but EMB allocation begins after the first HMA.

For example, suppose that you have a 4 megabyte system of which 3 of the 4 megabytes are available as extended memory and no memory has been assigned yet. Calling function 08H returns the number 3072 to both the AX register and the DX register for 3 megabytes. Then the XMM would be able to provide 3 megabytes of extended memory if you later called function 09H, as long as the memory is allocated after the HMA. The last 64K of this area exceed the end of extended memory, so 64K isn't physically available. Always subtract 64K from the value returned by function 08H. This deduction lets the size of the HMA govern the calculation.

Once you've determined the size of the largest available EMB, function 09H lets you request a corresponding amount of extended memory. Place the function number (09H) in the AH register, and the size of the desired block (in kilobytes) in the DX register. The DX register returns the handle needed for addition access to this EMB, unless an error occurred during memory allocation.

Check the AX register for error codes. A memory block allocation may fail, even if enough extended memory is available. The handles returned in the DX register may cause this failure. Since the XMM has only a limited number of these handles available, you may find that free extended memory exists, but all handles have already been assigned to many small EMBs. To prevent this condition, request large EMBs and try to subdivide them internally into various memory areas that are used for different purposes.

We've mentioned the XMM handles several times. You may be wondering why the XMM uses handles, instead of immediately returning the address of the allocated area. The XMM's memory management system was designed to avoid fragmentation of extended memory. This means the XMM will move the various EMBs back and forth in memory, creating a large memory range from smaller EMBs. This large EMB can then be passed to a program that calls function 09H. Another large area can be created from several small and released EMBs. This area can then be passed, as a whole, to a caller of function 09H. The addresses of the already allocated EMBs also change, so the XMM assigns logical handles instead of a physical address. This handle must be stored safely, since it's required for all future access to an EMB. If the EMB is released, enlarged, or at least partially copied into conventional memory, this handle is required by the XMM to identify an EMB.

Releasing EMBs

It's very easy to release an EMB to make it accessible to other programs. This release doesn't occur automatically at the end of a program because TSR programs can also use EMBs for their own purposes. Function 0AH releases allocated EMBs before terminating a program. If you don't pass this function, the program will continue to access the EMBs until you reset the computer. Place the function number (0AH) in the AH register and the EMB handle's number in the DX register.

Copying EMBs

Since data can be stored in extended memory but not manipulated, complete or partial EMBs must frequently be copied from extended memory into conventional memory, or from conventional memory to extended memory. Function 0BH performs this task. Place the function number (0BH) to the AH register, and a pointer to the extended move structure to the DS:SI register pair. This structure contains all the information needed about the source and target areas, as well as the number of bytes to be copied.

Conveying the handle and offset address information of the two areas must be handled differently, depending on whether the area is in extended memory or conventional memory. If you want to address an EMB, you must then indicate the handle that was returned by function 09H when allocating the EMB. The offset address represents the offset relative to the beginning of the block. However, if you want to address an area in conventional memory, the value 0 must be passed as the handle, and the segment and offset addresses of the beginning of the block must be passed as the offset. Remember the block's address must be passed in the order OFFSET:SEGMENT.

The execution speed of this copy process is slow, especially on 80286 based machines. There's no way to avoid the long and involved switching back from protected mode. The XMS standard has four additional EMB functions that are rarely used. You'll find information about these functions in the Appendices on the companion CD-ROM.

HMA access

XMM also provides access to the HMA. This minimizes memory conflicts between programs and offers hardware independent control of the A20 address line. This is important because not all PC systems control this line through a bit in the keyboard controller's output port, as we described in the previous section.

Unlike extended memory, HMA access cannot be parceled out to several programs because of its size. If a program requests access rights to the HMA using function 01H, it's an all-or-nothing situation. The program either receives complete access to the HMA, or no access at all. The latter occurs mainly when the HMA was already assigned, or when the size of the required HMA memory is less than the /HMAMIN parameter (an option available when the driver is initially called). This should prevent a TSR program (which may be only a few kilobytes in length) from receiving exclusive access rights to the HMA, while another program is denied access.

For normal purposes, pass FFFFH to the DX register as the amount of HMA memory needed when calling function 02H. For TSR programs, pass the exact number of bytes needed for the TSR to the DX register. The contents of the AX register indicate

whether the access rights were granted. The value 0001H indicates the release of the HMA, while the value 0000H indicates unsuccessful execution. The A20 address line must be switched before direct access to HMA can occur. The XMM provides a total of four different functions for enabling and disabling this line. Pass the function number to the AH register for calling.

The four functions enable and disable the A20 address line both locally and globally. This distinction is important; global functions always act on the A20 address line, but the local functions rely on an internal counter. This ensures that A20 access occurs only when the calls are balanced (enable/disable). After two consecutive calls of the local enable function (function 05H), the line switches off only if two local disable calls (function 06H) follow.

Displaying and hiding the mouse cursor involves a similar process (see Chapter 11 for more information). Except on rare occasions, you'll use the global functions in HMA access. Before ending program execution, remember to disable the A20 address using function 04H, and to release the HMA proper using function 04H. If you don't perform these two functions, programs called later will either crash (because a segment overflow was expected at the 1 megabyte limit) or will be denied access to the HMA.

Upper Memory Blocks

Upper Memory Blocks (UMBs) are within the processor's address space, which makes them directly accessible to a program. Unfortunately, RAM seldom exists in the range from 640K to 1 megabyte, unless you have an 80386 computer, in which programs are stored in upper RAM through software.

If the system for which you're developing software has UMBs available, or you just want to keep this memory option open, the XMS standard supports two functions for access to upper memory. MS-DOS Version 5.0 and up support UMB access, although this support merely acts as moderator between a program and the XMS driver (see Chapter 23 for more information).

Unlike the HMA, Upper Memory Blocks provide true memory management, which permits several programs to reside in upper memory. Segment addresses of the allocated blocks are used instead of handles. These segment addresses act as references for identifying a UMB for the XMM.

Allocating UMBs

Function 10H allocates a UMB when possible. Place the function number (10H) in the AH register, and the size of the block you want allocated (in paragraphs [= 16 bytes]) in the DX register. If enough memory isn't available, the DX register returns the size (in paragraphs) of the largest available UMB.

Releasing UMBs

The XMM lets you release a UMB as well as allocate one. Function 11H performs the release. Pass the function number (11H) to the AH register and the segment address of the addressed UMB to the DX register.

Demonstration programs

The following programs present some aspects of the XMS standard. Since these programs contain all the necessary XMS calls, you can adapt these routines to your own needs. Because XMM access is difficult in high level languages, both the Pascal and C implementations include some assembly language programming. C and Pascal libraries include compiler routines for calling almost any interrupt functions, but they don't include provisions for calling and passing register contents to normal subroutines.

The C implementation uses an assembly language program (XMSCA.ASM) and the Pascal implementation directly stores the assembly language as INLINE commands. This routine is similar to commands for calling interrupts. A structure is passed to call an XMM function. This structure contains the required processor register contents. After the function call the structure receives the return values. The main program checks for the availability of the XMM driver. If this driver exists, two additional procedures check the viability of the HMA and extended memory.


You'll find the following program(s) on the companion CD-ROM



XMSC.C (C listing)
XMSP.PAS (Pascal listing)
XMSCA.ASM (Assembler listing)

13

Sound On Your PC

 Every PC has a built-in speaker that beeps when certain errors occur or when the keyboard buffer is full. The speaker can also generate other sounds. This chapter demonstrates sound generation through software.

How the PC generates sound

Tones occur when the cone of a speaker *oscillates* (moves back and forth). A single oscillation creates a click instead of a musical sound. If a group of oscillations occur in rapid succession, a tone is produced. The pitch; (the note value) of a tone depends on the number of cycles (oscillations) that occur per second. The pitch of a tone in cycles per second is measured in Hertz. For example, if the speaker oscillates at a rate of 440 times per second, it generates a tone with a frequency of 440 Hertz. Certain pitches have specific note names assigned to them, such as A440 (the note that sounds at 440 Hertz). The following tables show the pitches and frequencies of tones generated by the PC. This range covers 8 octaves (almost the range of a full piano keyboard):

Octave 0		Octave 1		Octave 2		Octave 3	
C	16.35	C	32.70	C	65.41	C	130.81
C#	17.32	C#	34.65	C#	69.30	C#	138.59
D	18.35	D	36.71	D	73.42	D	146.83
D#	19.45	D#	38.89	D#	77.78	D#	155.56
E	20.60	E	41.20	E	82.41	E	164.81
F	21.83	F	43.65	F	87.31	F	174.61
F#	23.12	F#	46.25	F#	92.50	F#	185.00
G	24.50	G	49.00	G	98.00	G	196.00
G#	25.96	G#	51.91	G#	103.83	G#	207.65
A	27.50	A	55.00	A	110.00	A	220.00
A#	29.14	A#	58.27	A#	116.54	A#	233.08
B	30.87	B	61.74	B	123.47	B	246.94

Octave 0		Octave 1		Octave 2		Octave 3	
C	261.63	C	523.25	C	1046.50	C	2093.00
C#	277.18	C#	554.37	C#	1108.74	C#	2217.46
D	293.66	D	587.33	D	1174.66	D	2349.32
D#	311.13	D#	622.25	D#	1244.51	D#	2489.02
E	329.63	E	659.26	E	1328.51	E	2637.02
F	349.23	F	698.46	F	1396.91	F	2793.83
F#	369.99	F#	739.99	F#	1479.98	F#	2959.96
G	392.00	G	783.99	G	1567.98	G	3135.96
G#	415.30	G#	830.61	G#	1661.22	G#	3322.44
A	440.00	A	880.00	A	1760.00	A	3520.00
A#	466.16	A#	923.33	A#	1864.66	A#	3729.31
B	493.88	B	987.77	B	1975.53	B	3951.07

The speaker in the PC can generate frequencies from 1 Hertz up to more than 1,000,000 Hertz. However, most human ears are only capable of hearing frequencies between 20 and 20,000 Hertz. Also, PC speakers don't reproduce music very well because they play some tones louder than others. Since the speaker has no volume control, this effect cannot be changed.

A sound program should oscillate the speaker according to the frequency of the tones desired. The following is a rough outline of a possible sound generation program:

- Execute the instruction to move the cone forward, then undo the instruction (i.e., move the cone back to its original position). Repeat these steps in a loop so that it occurs as many times per second as required by the frequency of the tone being generated.

The previous procedure has several disadvantages:

- The execution speed of individual instructions depends on the processing speed of the computer.
- This program must be adjusted to the processing speed of individual computers.
- The tone becomes distorted when the tone production loop ends.

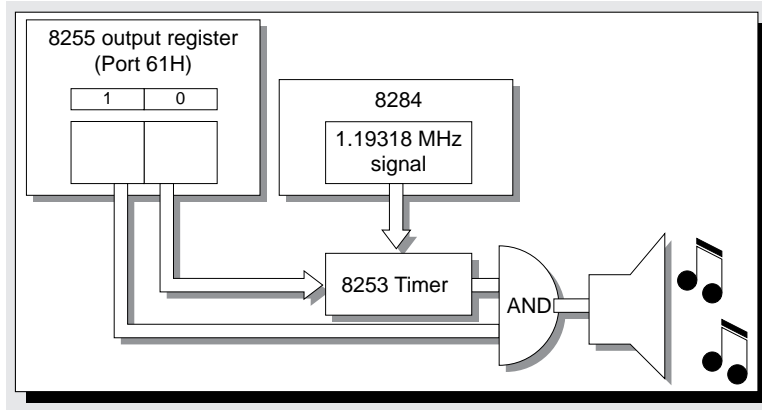
8253 timer

Every PC uses one particular chip for tone generation: The 8253 programmable timer, which actually maintains control of the internal clock. The 8253 can perform both timing and sound because of its ability to enable a certain action at a certain point in time. It senses timing from oscillations it receives from the PC's 8284 oscillator, which generates 1,193,180 impulses per second. The 8253 can then be instructed how many of these impulses it should wait before triggering a certain action. In the case of tone generation, this action consists of sending an impulse to the speaker. Before executing this action, the chip must be programmed for the particular frequency it should generate. The frequency must be converted from cycles per second into the number of oscillations coming from the oscillator. This is done with the help of the following formula:

$$\text{counter} = 1,193,180 / \text{frequency}$$

The result of this formula, the variable counter, is passed to the chip. As the formula demonstrates, the result for a high frequency is relatively low, and the result for a low frequency is relatively high. This makes sense because it tells the 8253 chip how many of the 1,193,180 cycles per second it must wait until it can send another signal to the speaker. The lower the value, the more often it sends a signal to move the speaker cone back and forth, which produces a higher tone.

Creating a tone on the PC

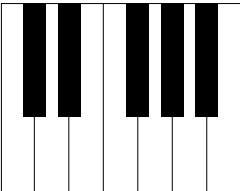
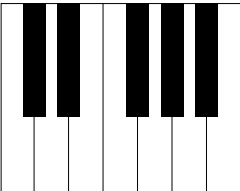
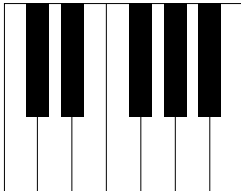


Ports and PC sound

Communication between the CPU and the 8253 chip occurs through ports. First, the value 182 is sent to port 43H. This instructs the 8253 that it should start generating a signal as soon as the interval between individual signals has been passed. This interval is the value that was calculated with the previous formula. Since the 8253 stores this value internally as a 16-bit number (a value between 0 and 65,535), it limits the range of tones generated to frequencies between 18 and 1,193,180 Hertz. This number must be transmitted to port 42H. Since this is an 8-bit port, the 16 bits of this number cannot be transmitted simultaneously. First the least significant eight bits are transmitted, then the most significant eight bits are transmitted.

Now the second step occurs: The 8253 signal is sent to the speaker. The speaker access occurs through port 61H, which is connected to a programmable peripheral chip. The two lowest bits of this port must be set to 1 to transmit the 8253 signal to the speaker. Since the remaining six bits are used for other purposes, they cannot be changed. For this reason, the contents of port 61H must be read, the lowest two bits must be set to 1 (an OR combination with 3), and the resulting value must be returned to port 61H. A tone sounds, which ends only when the bits that were just set to 1 are reset again to 0.

Keyboard setup and timer frequencies

Octave 3							Octave 4							Octave 5						
C	#D	#	F	#G	#A	#	C	#D	#	F	#G	#A	#	C	#D	#	F	#G	#A	#
																				
C	D	E	F	G	A	B	C	D	E	F	G	A	B	C	D	E	F	G	A	B
<hr/>																				
C = 9121	F# = 9121						C = 4560	F# = 3224						C = 2280	F# = 1612					
C# = 8609	G = 8609						C# = 4304	G = 3043						C# = 2152	G = 1521					
D = 8126	G# = 8126						D = 4063	G# = 2873						D = 2031	G# = 1436					
D# = 7670	A = 7670						D# = 3834	A = 2711						D# = 1917	A = 1355					
E = 7239	A# = 7239						E = 3619	A# = 2559						E = 1809	A# = 1292					
F = 6833	B = 6833						F = 3416	B = 2415						F = 1715	B = 1207					

Sample programs

GW-BASIC and Turbo Pascal contain resident sound commands. The machine language programmer and C programmer must create their own sound applications.

This chapter contains demonstration programs for both of these languages. These programs can be added to your own C or assembly language programs.

How they work

Both programs produce tones for specific time periods. This is done with the help of the timer interrupt 1CH, which is called by the timer interrupt 8H 18.2 times per second. When the tone generation routine executes, it receives the frequency of the tone and the tone's duration (length). The duration is measured in 18ths of a second, so the value 18 corresponds to a second and the value 9 corresponds to a half-second. This value is stored in a variable.

Immediately before activating the tone output, the interrupt routine of interrupt 1CH turns to a user-defined routine. This routine, called 18.2 times per second, decrements the tone duration in the variable during every call. When it reaches the value, the tone duration ends and the tone must be switched off. The routine allocates a variable to notify the actual sound routine of this end. The sound routine recognizes this immediately because it has been in a constant wait loop since switching on the tone. This loop simply monitors the contents of this variable. After recognizing the end of the tone, it stops the sound output and returns the timer interrupt to its old routine.

The sound routine requires the number assigned to this tone rather than the frequency itself. This number is related to the table containing the frequencies of octaves 3 to 5. The value 0 represents C of the third octave, 1 represents C-sharp, 2 represents D, 3 represents D-sharp, etc.

Both the C program and assembly language program demonstrate the sound routine by playing a scale over the course of two octaves, with each note sounding for a half a second. The machine language demo program and the sound routine are stored in one file. The C versions of these programs are split into two source code files. The C demo program contains the sound function call only, and the machine language program, which creates the sound, must be linked to the demonstration program.

You'll find the following program(s) on the companion CD-ROM



SOUNDA.ASM (Assembler listing)
SOUNDC.C (C listing)
SOUNDCA.ASM (Assembler listing)

14

Diskettes And Hard Drives

The main purpose of the BIOS routines is to perform low-level functions on behalf of DOS. For example, BIOS routines can physically format the surface of a floppy diskette or access the sectors of a hard drive. DOS, however, remains the master controller for these processes. Most applications perform disk drive operations at the DOS level instead of the BIOS level. However, there are exceptions. For example, disk utilities, such as PC Tools or Norton Utilities, access the disk drive at the BIOS level. But generally these specialized programs are rare.

In this chapter we'll show you how to access the disk drives using the BIOS functions. Since all disk controllers aren't programmed identically, we won't be programming the disk controller directly. Most of the functions that you can perform at the disk controller level can also be performed at the BIOS level. It's worth using the BIOS functions to avoid hardware-dependent disk controller programming.

In addition to BIOS functions, we'll also discuss a few topics related to the hard drive. We'll explain the different types of hard drive controllers and see how hard drives record data.

We'll end the chapter with a discussion of hard drive partitioning, which lets the user divide the hard drive into several logical drives.

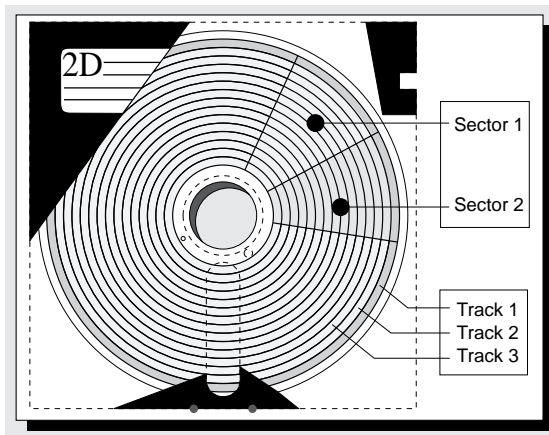
Floppy Diskette And Hard Drive Structure

Let's begin with a look at the common characteristics of floppy diskettes and hard drives. Floppy diskettes and hard drives have similar structures, which is indicated by the number of BIOS functions that apply to both. If you think of a floppy diskette as a two-dimensional version of a hard drive, the similarities are even more apparent.

Floppy diskette structure

A floppy diskette consists of individual tracks arranged as concentric circles at equal intervals over the surface of the diskette's magnetic media. These tracks are labeled from 0 to N; N represents the total number of sectors minus 1 and varies depending on the format. The outermost track is always numbered 0, the next track is numbered 1, etc. This process continues to the innermost track.

Structure of a 5.25-inch diskette



Each track is subdivided into a fixed number of sectors. Each sector holds the same amount of data. Sectors are numbered from 1 to N; N represents the number of sectors per track. The maximum number of sectors in a track depends on the type of floppy disk drive and the diskette's format. Each sector contains 512 bytes and is the smallest amount of data that a program can access. In other words, you must read or write a complete sector at a time. It isn't possible to read or write a single byte from the diskette.

The data in each sector is recorded using either an FM or MFM technique. These are the same recording methods used in hard drives (refer to the "Recording Information On The Hard Drive" section for more information). As a programmer, you don't have to worry about these details to access the data from the disk drive.

Use the following formula to calculate the capacity of a floppy diskette. Remember, this formula is for a single side of a diskette. If the floppy disk drive has two read/write heads (like most recent floppy drives), you must double this value. DOS refers to these sides of a diskette as side 0 and side 1.

```
Sectors * tracks_per_sector * 512 [bytes per sector]
```

The number of sectors per track also affects the data transfer rate. The data transfer rate is the speed of the floppy disk drive electronics and its controller. With a constant rotation speed of 300 revolutions a minute, the more bits per time unit that pass by the read/write head, and the more sectors can be written to a track.

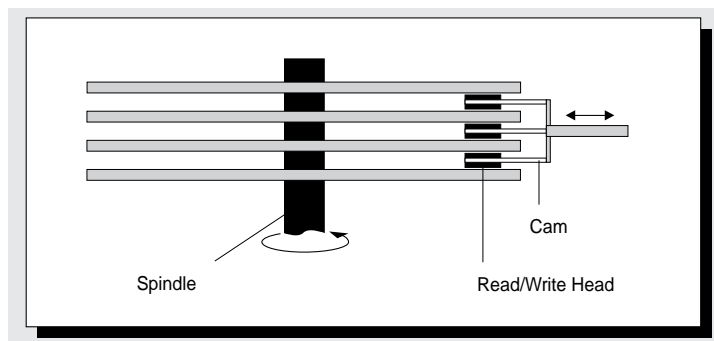
Hard drive structure

Since a hard drive rotates ten times faster than a floppy diskette, its data transfer rate is at least ten times higher than a floppy diskette's. The data transfer rate increases by a second power of ten, because modern 3.5-inch hard drives can store almost 100 sectors per track.

These characteristics don't change the fundamental structure of a hard drive. Think of a hard drive as a group of magnetic plates stacked on top of each other. Each magnetic plate is similar to a floppy diskette; it has two sides, is divided into tracks, and each track is subdivided into sectors. Above the surface of each side of the plate is a read/write head that accesses the data. The plates are aligned so track 0 on one of the plates is exactly above track 0 of another plate.

A read/write arm links all the read/write heads together. To access a particular track on one of the plates, the arm moves all the read/write heads to the specific track. Since this arrangement requires only a single positioning mechanism (the read/write arm), it simplifies the design and lowers the cost of the hard drive. However, with this arrangement, all the read/write heads must be moved to access data on a different track. So, to read data on track 1 of one plate, then data on track 50 of a different plate, and finally data on track 1 of the first plate again, the entire read/write arm must be moved twice. Positioning the arm like this requires a significant amount of time compared to the data transfer time.

Hard drive structure



To minimize the time needed to access data, you should prevent the data from being spread across multiple tracks. One way to optimize access time to a group of data is to write that data sequentially on a single track. If the data doesn't fit on a single track, then write in on the same track of a different plate. By doing this, the read/write arm doesn't need to be moved. Instead,

only the appropriate read/write head needs to be selected to read the desired data. Selecting (changing) heads is much faster than physically moving a mechanical read/write arm to change tracks.

The term cylinder is used to describe the multiple plates stacked on top of each other. A cylinder refers to all tracks that have the same track number but are located on different disk plates.

Disk Drives And Diskette Formats

Before describing the diskette BIOS routines, let's review the various diskette formats. Today's powerful PCs feature at least one of the following diskette formats:

1. 3.5-inch, 1.44 Meg high-density (HD)
2. 3.5-inch Super high density
3. 5.25-inch, 1.2 Meg high-density (HD)

Many sizes and formats have been used over the years (maybe you heard of the 8-inch floppy diskette?). The 3.5-inch, 1.44 Meg high-density (HD) format is currently the more popular of the three formats listed above. Its dominance is likely to continue for some time.

Diskette formats

Diskettes require formats that specify the exact qualities of a volume. Non-DOS diskettes cannot be read by DOS unless a special device driver is available. The following tables show the formats available for 3.5-inch and 5.25-inch diskettes:

3.5-inch diskettes formats					
Type	Density	Capacity	Tracks	Sectors	DOS Version
DS DD	135 tpi	720K	80	9	3.2
DS DD	135 tpi	1.44 Meg	80	18	3.3
DS DD	270 tpi	2.88 Meg	80	36	3.3

5.25-inch diskettes formats					
Type	Density	Capacity	Tracks	Sectors	DOS Version
SS SD	40 tpi	160K	40	8	1.0
SS SD	40 tpi	100K	40	9	2.0
DS SD	40 tpi	320K	40	8	1.1
DS SD	40 tpi	360K	40	9	2.0
DS HD	96 tpi	1.2 Meg	80	15	3.0

5.25-inch floppy disk drives and diskettes

The 5.25-inch floppy disk drive became the first PC standard floppy. Its still standard equipment for many PCs. Original 5.25-inch diskettes were single density diskette with four sectors per track and 40 sectors on each side. This provided a capacity of 80K per side or 160K for floppy disk drives with two read/write heads. Its transfer rate of 125K/sec is agonizingly slow compared to today's standards.

The single density diskettes were followed by double density diskettes. These diskettes doubled the number of sectors per track to eight while retaining the number of tracks per side (40). So, the capacity of the diskette is increased to 160K for a single-sided floppy disk drive and 320K for a double-sided floppy disk drive. The data transfer rate also doubled (to 250K/

sec). This 320K format was short-lived. Eight sectors doesn't quite fill up a track on double density diskette. So, there is still room for a ninth sector. Adding an extra sector to each track increases the capacity to 360K.

When the IBM/AT computer was introduced, a new format was also introduced. The new format, called high density diskette, gives the 5-1/4" floppy diskette a capacity of 1.2 Meg. The number of sectors per track was increased to 15 and the number of tracks per side was doubled to 80. As with double density diskettes, both sides of the diskette are used for this high density format. Theoretically it's possible to have 16 sectors on each track. But for practical considerations, developers settled on the 15 sectors per track arrangement to ensure a reliable floppy disk drive. High capacity floppy disk drives rotate at 360 RPM.

Instead of the 40 tracks used by the double density format, the high density format squeezes 80 tracks onto a diskette. So standard double density floppy disk drives cannot read or write this format. Only newer model MF (multifunction) drives are capable of reading and writing this format. A MF drive also adjusts to the standard double density format, which makes it possible to read and write standard 360K diskettes. In MF drives, the data transfer rate can be adjusted and the rotational speed can be reduced to the normal speed of 300 revolutions a minute. However, the higher number of sectors per track means that earlier PC and XT floppy disk drives cannot read or write to these high density diskettes. These drives cannot achieve the necessary data transfer rate of 500 kilobits per second because they cannot be designed or configured for the task.

Increasing the recording capacity from double density to high density isn't simply a matter of drive electronics. It's also a question of the "granulation" of the magnetic material on the diskette. The smaller the single magnetic particles, the more information can be recorded on a given surface. This is also the reason why double density diskettes can never be formatted error free in AT disk drives at 1.2 Meg; the granulation on those diskettes is simply too coarse.

While it may be possible to format a double density diskette at high density, these diskettes can usually be read only on the PC on which they were formatted. Other PCs will simply give you read errors, making it almost impossible to use the diskette. This is caused by the variances in the positioning of the read/write heads on different floppy disk drives.

The differences between read/write head positioning may only amount to fractions of a millimeter. A single track may actually be wider than the read/write head, allowing the head to be positioned within the track's range. Even allowing for variances in track size, this allows the head to correctly read the information. However, this causes problems when reformatting a standard double density diskette using a high density drive, because the high density drive's formatting capabilities may cause problems when a PC/XT double density drive attempts to read the newly formatted disks. The smaller read/write tolerances may cause track skipping and read errors.

Something similar occurs if you try to format a high density diskette in a double density floppy disk drive. Usually this is also doomed to failure or results in read errors on other drives. So, purchasing expensive HD (High Density) diskettes for your PC or XT disk drives isn't necessarily a good idea.

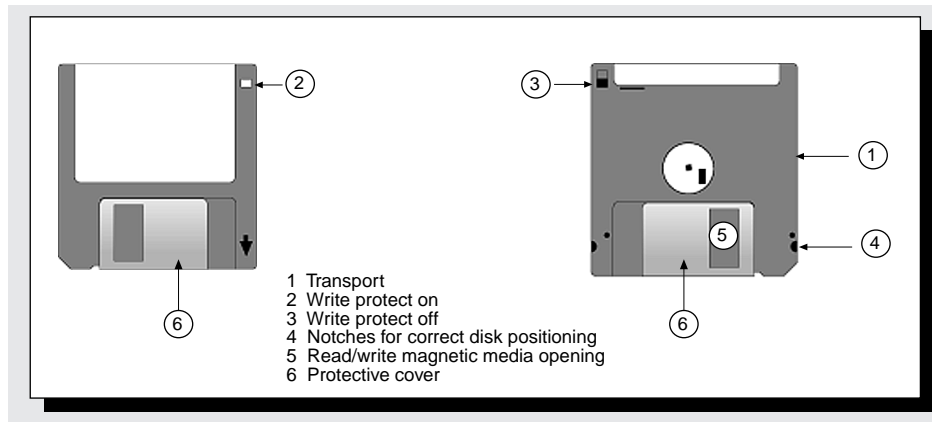
3.5-inch floppy disk drives and diskettes

Although 5.25-inch floppy disk drives are still widely used, they are being replaced by the smaller 3.25" floppy disk drives. These smaller drives first became popular on the laptop computers. Now, 3.5-inch floppy disk drives are also used on most desktop PCs. The 3.5-inch diskettes are preferred because of their convenient size and the sturdiness of their rigid plastic case. The magnetic surface of a 3.5-inch diskette is covered by a sliding metal door to protect the data from damage by dust and dirt particles.

The 3.5-inch floppy disk drives record data in either double density or high density. Double density has a capacity of 720K, with 9 sectors on a track and 80 tracks on each side of the diskette. A 3.5-inch diskette has a considerably higher track density than a 5.25-inch diskette, especially since the diameter of the magnetic media is much smaller.

The original 3.5-inch floppy disk drives have a double density format with a capacity of 720K. The newer 3.5-inch floppy disk drives have a 1.44 Meg capacity. A 1.44 Meg floppy disk drive is now the standard in 3.5-inch drives today. It also has 80 tracks per side, but has 18 sectors per track. This also doubles the data transfer rate, which makes it impossible to use these diskettes in the double density 3.5-inch floppy disk drives of earlier PCs and XTs.

Structure of a 3.5-inch diskette



A 3.5-inch high density floppy disk drive performs like a 5.25-inch MF floppy disk drive in that it can adjust to reading, writing and formatting double density diskettes.

Recently, a new 3.5-inch format was introduced. Diskettes using this format have a capacity of 2.88 Meg. This format is called extra high density (ED). ED diskettes have double the number of sectors (36) and can only be read by the new ED floppy disk drives. Like HD drives, ED floppy disk drives are also downwardly compatible. This means they are able to process both double density and high density diskette formats.

Floppy disk drives that can read and write the different formats must be able to determine the format in which the diskette is written. Then it can pass this information to the BIOS before accessing the data on the diskette. Finding the format of a 5.25-inch diskette isn't easy, because this information can be determined only by reading the data. So, the diskette must already be formatted.

However, the capacity of a 3.5-inch diskette can be determined by a small hole, which is located on the opposite side of the write-protect slider. Within the floppy disk drive itself is a light-sensitive sensor that can detect the presence or absence of the hole. Unlike high density diskettes, double density diskettes don't have this hole. Extra high density diskettes also have a hole, but the hole is located in a different position.

Disk drives and their controllers

A floppy disk drive consists of a motor that rotates the diskette at 300 revolutions per minute (360 RPM for HD 5.25-inch) and a mechanism for moving the read/write head. The drive also has an electronic component, called a data separator. The data separator converts a voltage into a binary data stream as the read/write head passes over the surface of the diskette.

The floppy drive is controlled by a separate diskette controller, which is either part of the computer's motherboard or on an I/O card in one of the computer's expansion slots. The main functions of the diskette controller are performed by an NEC PD765 or a similar chip from another manufacturer. Only NEC chips or NEC compatible chips are able to work with the ROM-BIOS. After all, it's the ROM-BIOS that uses this chip to control access to the floppy disk drive. Although it's possible to adapt the ROM-BIOS to work with another diskette controller, most manufacturers of ROM-BIOS systems use the established standard of the NEC PD765.

However, this standard causes problems with the new ED floppy disk drives. These drives have twice the track capacity (36 instead of 18 sectors) and the twice the data transfer rate. But both parameters are unknown to the BIOS. A ROM extension on the controller card can be used to avoid these limitations. The ROM extension wedges itself into the ROM-BIOS when DOS is first started. Then it manages all access to these floppy disk drives. Since the ROM extension is intended to be used only in real mode, it cannot be used if the computer is running protected mode.

Operating systems such as UNIX or OS/2 run in protected mode. These systems rely on the diskette controller to handle all the details of accessing the drives. So, they'll fail with new floppy disk drives unless special device drivers are written into

the operating system. Hopefully a BIOS standard for the new ED drives will be developed soon, making ROM extensions unnecessary. Since Windows will be taking over more of the BIOS tasks in the future, this problem will become more prevalent for PC users.

Accessing Floppy Disk Drives With The BIOS

There is a complete set of BIOS functions that access floppy disk drives. Interrupt 13H is used to call these functions. This interrupt is also an interface to the hard drive utilities of the BIOS. Wherever possible, similar floppy and hard drive functions share an identical function number. To differentiate between the drives, the drive specification is passed to the function in the DL register.

For floppy disk drives, either the value 0 (for drive A:) or 1 (for drive B:) is used. A few disk controllers support four floppy drives by providing a BIOS extension that also accepts the values 2 and 3 for the other two floppy disk drives. Hard drives are specified by the values 80H and 81H.

Here also, you must distinguish between the PC/XT-BIOS and the AT-BIOS. For example, there are a few BIOS functions that are specific to MF floppy disk drives. The following table lists the diskette functions of the BIOS interrupt 13H:

Diskette functions of the BIOS interrupt 13H							
No.	Tasks	PC/XT	AT	No.	Tasks	PC/XT	AT
00H	Reset	Yes	Yes	08H	Request Format	Yes	Yes
01H	Read status	Yes	Yes	15H	Define drive type	No	Yes
02H	Read	Yes	Yes	16H	Detect diskette change	No	Yes
03H	Write	Yes	Yes	17H	Determine diskette format	No	Yes
04H	Verify	Yes	Yes	18H	Determine diskette format	No	Yes
05H	Format	Yes	Yes				

Notice functions 17H and 18H have the same task. This isn't an error. Function 18H was introduced for the 3.5-inch HD drives. The older function 17H isn't capable of supporting the drive, so it was replaced by a new function. We'll discuss this in more detail later.

Drive status

These BIOS functions also have another similarity. They return a status or error code. This status code is returned to the caller in the AH register. A nonzero value and a set carry flag indicate an error.

Status and error codes of the BIOS diskette functions			
Code	Meaning	Code	Meaning
00H	No error	08H	DMA overflow
01H	Illegal function number	09H	Data transfer past the segment limit
02H	Address marking not found	10H	Read error
03H	Attempt to write to write-protected diskette	20H	Diskette controller error
04H	Addressed sector not found	40H	Track not found
06H	Diskette was changed	80H	Time out error, drives does not respond

You can also determine the diskette status through function 01H. Simply pass function number 01H in the AH register and the drive specification value in the DL register. After the function call, the drive status is returned to you in the AH register.

Resetting the floppy disk drive

After an error you must reset the floppy disk drive. To do this, use function 00H. Pass function number 00H in the AH register and the drive specification value in the DL register. After the function call, the current drive status is returned to you in the AH register. It doesn't matter whether you pass 0 or 1 as the drive specification; all the floppy disk drives will be reset. Remember the value in the DL register isn't ignored. Entering a value greater than 80H resets the hard drives.

Prompt for the drive type

A program needs to know the type and format of a floppy disk drive to use it. You can use the 08H and 15H functions to determine this information. Function 08H, which is found in the PC/XT-BIOS, is used to distinguish between the different floppy and diskette formats. Pass function number 08H in the AH register and the drive specification in the DL register. The following illustration shows the information that's returned.

Any error code is returned in the AH register with the carry flag set. By using this function, you can determine whether a given drive is installed to demonstrate, for example, that a second, third, or even fourth drive is present.

The value in the BL register is especially important. This value not only reveals the floppy disk drive type (3.5-inch or 5.25-inch), but also shows the diskette format (DD or HD). However, this value doesn't necessarily describe the format of the diskette that is located in the drive; it describes only the highest possible density. This information is taken from the CMOS-RAM, in which this information is stored when the computer is originally setup. The Drive Type value is standardized, but doesn't yet include the new ED 3.5-inch drives. However, in the future these new types will most likely appear under Drive Type 05H. Although the number of sectors, tracks, and heads can be derived from the Drive Type value, this information is also specified explicitly in the CH/CL, DH/DL registers.

Information returned by the 08H function	
Register	Information
BL	Drive Type 01H = 5.25-inch, 360K 02H = 5.25-inch, 1.2 Meg 03H = 3.5-inch, 720K 04H = 3.5-inch, 1.44 Meg
DH	Maximum number of sides (always 1)
CH	Maximum number of tracks
CL	Maximum number of sectors
ES:DI	Pointers to DDPT

The DDPT, which is referenced by the pointer in the register pair ES:DI, is the Disk Drive Parameter Table. This table contains a parameter the BIOS needs for programming the diskette controller. You'll find a description of this table later on in this chapter.

Function 15H has a different purpose. This function is only supported by ATs and their FM drives. Unlike PC/XT floppy disk drives, these drives are able to detect when a diskette has been changed. This feature is important for programs that depend on the presence of a specific diskette. This is true especially for DOS, which reads the FAT table before it accesses a diskette to determine which sectors of the diskette are occupied and which sectors are still unused.

If the diskette is changed without DOS knowing about it, DOS may continue to use the contents of the original diskette's allocation table (FAT) and may inadvertently overwrite and/or destroy data on the newly inserted diskette. However, if the same diskette is reinserted into the floppy, DOS won't have to reread the FAT again.

Drive codes of function 15H	
Code	Meaning
AH = 00H	Drive not present
AH = 01H	Disk drive, does not recognize diskette changes
AH = 02H	Disk drive, recognizes diskette changes
AH = 03H	Hard drive

Function 15H of the BIOS helps determine if a diskette has been replaced. Pass function number 15H in the AH register and the drive specification in the DL register. The table on the left shows what information is returned after the function call.

Reading diskette sectors

Reading diskette sectors is one of the most basic tasks the BIOS performs. Function 02H reads diskette sectors. Remember that this function can read several sectors through a single call if they're on same track and contiguous. Remember the data isn't transferred to a fixed memory location. Instead, the address of a buffer is passed in the register pair ES:BX. Register ES contains the segment address of the buffer and register BX contains the buffer's offset address.

Register when calling function 02H	
Register	Information
AL	Number of sectors to be read
DL	Drive specification value
DH	Side (0 or 1)
CL	Sector number (1 to N)
CH	Track number (0 to N-1)
ES:BX	Address of the buffer for the data to be read
AH = 03H	Hard drive

After the function call, the error status is returned in the AH register and the number of read sectors read is returned in the AL register. If the carry flag is set, it signals an error.

If you're using an MF drive, you can use a trick to determine the format of a diskette. By trying to read a diskette with a sector greater than 9, you can determine whether a diskette is DD or HD. Since the maximum number of sectors per track on a DD diskette is 9, function 15H will return a disk status error. The track number isn't important, but it should be less than 40. If a disk status error is returned, don't immediately abort the operation. You should repeat all read, write, and format operations at least three times before you give up and assume there is a "real" error. Often an operation fails the first time, but succeeds when you try it a second or third time. Perhaps the read/write head wasn't positioned properly the first time or the floppy drive wasn't synchronized to the electronics yet (see the "Recording Information On The Hard Drive" section in this chapter).

In cases of errors, you don't have to worry about the validity of the data because the drive uses parity checking to ensure the data in each sector is correct.

Writing diskette sectors

Function 03H is used to write to individual sectors. The parameters are passed according to the table on the right. Remember the buffer, to which the ES:BX register pair points, must contain the data to be written to the diskette.

Register when calling function 03H	
Register	Information
AL	Number of sectors to be written
DL	Drive specification value
DH	Side (0 or 1)
CL	Sector number (1 to N)
CH	Track number (0 to N-1)
AL	Number of sectors to be read
ES:BX	Pointer to the buffer containing the data

Verifying diskette sectors

Function 04H tests whether data have been correctly transferred to the diskette. The data in the memory aren't compared with the data on the diskette. Instead, a CRC value is used to determine whether the data was transferred correctly. CRC, which is an abbreviation for "Cyclical Redundancy Check", is a very reliable procedure for verifying accuracy. This procedure combines the values of each byte within the sector with a checksum through a complicated mathematical formula.

Since most disk drives are very reliable, most programmers consider this routine to be unnecessary and don't use it. DOS uses this function when writing data only if the DOS VERIFY ON command is active. The parameters for function 04H are the same as for function 02H and 03H except that a buffer address isn't required.

Formatting individual tracks on a diskette

Function 07H is used to format an entire diskette. But it's also possible to format individual tracks on a diskette. To do this, first use function 18H to tell the BIOS which format to use. The function number 18H is passed in the AH register, the drive specification value in the DL register, the number of tracks in the CH register, and the number of sectors per track in the CL register. After the function call, the carry flag signals the specified format is supported by the floppy disk drive. In this case, the register pair ES:DI is a pointer to DDPT, which is required for subsequent formatting functions.

We'll describe the DDPT in more detail later. For now, remember the pointer must be passed to interrupt vector 1EH, in which the BIOS keeps a pointer to the current DDPT.

After function 07H sets the desired format and the DDPT pointer is passed to interrupt vector 1EH, you can start the actual formatting process. To do this, use function 05H. This function formats a complete track. Although you can format individual sectors with 128, 256, 512, or even 1024 bytes per sector, only a 512 byte format can be used under DOS. This is because DOS supports only this size sector.

Register when calling function 05H	
Register	Information
AL	Number of sectors in the track
DL	Number of the drive
CH	Number of the track
DH	Side (0 or 1)
ES:BX	Pointer to format table

To use function 05H, pass the drive specification value in the DL register, the diskette side in the DH register, the number of sectors per track in the AL register, and the track number in the CH register (0 through 39 or 0 through 79).

You'll see the ES:BX register pair points to a "format table". This table represents the formatting attributes. The table on the right shows this as an array of 4-byte entries (one for each sector to be formatted). Although the track number and diskette side is passed to function 05H, it must be repeated in the table. The sectors are physically created in the same sequence as the table entries. So it's possible to format the first entry as sector number 1 and the second entry as sector number 7. The logical sector number is recorded in the header of each sector on the diskette, so the floppy disk drive can later identify the sector that is being searched.

ES:BX register pair "format table"	
Offset	Meaning
0	Track to be formatted
1	Diskette side (always 0 for one-sided diskettes): 0 = Front side 1 = Back side
2	Number of the sector
3	Number of bytes in this sector: 0 = 128 bytes 1 = 256 bytes 2 = 512 bytes 3 = 1024 bytes

Since the BIOS doesn't define the logical sector numbers, you can change the interleaving. Generally only hard drives use interleaving, as we'll see in Section 6.7. Since interleaving doesn't have any advantages for floppy diskettes, you should number the sectors consecutively when you create the format table.

The number of bytes per sector don't have to be identical either, since these numbers are defined explicitly for each sector in the table. You can change the number of bytes per sector to develop a form of copy protection, for example. We'll soon see how this is done. A program at the end of this chapter shows how to format diskettes using functions 18H and 05H.

Disk Drive Parameter Table

To program the diskette controller, the BIOS needs the physical formatting information described above and some additional information. We've already introduced you to the Disk Drive Parameter Table (DDPT). The ROM BIOS contains a table for every drive and supported diskette format. Also, you can define your own DDPT, since the BIOS always references the current DDPT through a FAR pointer, which is contained in the memory locations in which the interrupt vector 1EH is usually found. Since neither DOS nor the PC hardware use interrupt 1EH, you can change the contents of these memory locations.

DOS creates its own DDPT. This DOS DDPT is designed to speed up access to the diskette. The table is 11 bytes in size, as shown by the table on the following page. Not all the parameters can be changed. However, the entries that are marked with an asterisk may be changed.

The first field of the DDPT table actually has two subfields: The step rate (bits 4-7) and the head unload time (bits 0-3). The step rate describes the time the controller has to move the read/write head from one track to another. This value is represented as milliseconds, with the value 0FH representing 1 ms, 0EH representing 2 ms, 0DH representing 3 ms, etc. The head unload time describes the time the read/write head has to lift up off of the surface of the diskette, for example when changing tracks. It's specified as a factor of 16 ms. The default value 0FH (240 ms) is extremely conservative and can usually be lowered.

Diskette functions of the BIOS interrupt 13H					
Offset	Meaning	Type	Offset	Meaning	Type
*00H	Step rate and head unload time	1 BYTE	06H	DTL (Data Length)	1 BYTE
*01H	Head load time	1 BYTE	07H	Length of GAP3 when formatting	1 BYTE
*02H	Post run-time of diskette motor	1 BYTE	*08H	Fill character for formatting	1 BYTE
03H	Sector size	1 BYTE	*09H	Head settle time	1 BYTE
04H	Sectors per track	1 BYTE	*0AH	Time to run up of diskette motor	1 BYTE
05H	Length of GAP3 when reading/writing	1 BYTE			

The second field is also two subfields: The head load time (bits 1-7) and the DMA flag (bit 0). The head load time is the time the read/write head has to settle to the surface of a track. This value is expressed as a factor of 2 ms. In accessing a diskette, it's usually necessary to wait much longer for the diskette motor to reach its required speed. So it's common to specify a very low value (1 or 2) for the head load time. The DMA flag is represented by bit 0. This flag must always be set to 0.

The third field is the post runtime of the diskette motor after a diskette operation. This is the period of time that elapses until the diskette motor is switched off when no other diskette operations are being performed. Because it takes a relatively long time to get the motor running, it shouldn't be switched off immediately after each diskette access. This value is related to a cycle of approximately 18 ticks per second (1 tick is approximately 55 ms). So a value of 18 represents a post runtime of about one second. The default value is 25H, which is approximately two seconds.

The fourth field specifies the number of bytes per sector that can be used in a read or write operation. This corresponds to the values for formatting a sector, so it usually contains the value 3 for 512 bytes per sector. To read or write to sectors with different sector sizes, you must first enter the appropriate value in this field.

The next field at offset address 04H is the maximum number of sectors per track, which depends on the selected diskette format. The next three fields refer to the coding and decoding of sector information, which is stored on the diskette along with the actual data. You should never tamper with these values. However, the field at offset address 08H can be changed. This field contains the ASCII code of the fill character to be used when the diskette is being formatted. During formatting, as the sectors are created, they are also given a fixed contents. The default fill character is a division sign (ASCII code 246).

The next field contains the head settle time. After the read/write head travels from one track to another, a short delay is needed to allow the vibrations from this movement to subside. Only then can the read/write head perform the subsequent data access properly. The value in this field represents a delay in milliseconds. The default value is 25 ms. The last field in the DDPT specifies the time it takes for the diskette motor to attain its operating speed. The value in this field is a factor of 1/8 seconds. While DOS defaults to a value of 1/4 second, the BIOS equivalent is 1/2 second.

Sample programs

Changing the various values in the DDPT won't produce performance miracles. However, you'll probably want to experiment with it. We've developed two small programs in Pascal and C to set the various parameters of the current DDPT. However, the programs won't change all the parameters because this would be too dangerous.

The programs DDPTP.PAS and DDPTC.C both work according to the same principle. You call both programs from the DOS command line without specifying any parameters. Each program displays the contents of the current DDPT (the DDPT of the last accessed disk drive). To address a certain disk drive, use the DIR command before the DDPTP or DDPTC program.

You can change the contents of a DDPT field by typing the command with the appropriate parameter. The parameter is a two-letter code (the code that appears on the screen when the fields are displayed), followed by a colon, and then followed by a two digit hexadecimal number that represents the new value.

For example, if you type the following command:

DDPTP MA:04 SR:08

the starting time of the diskette motor is set to one-half second and the step rate is lowered to 8 ms. This command will work only if the DDPT is in memory, not the one in the ROM-BIOS. Each program will indicate whether you're trying to change the contents of the ROM.

Do it yourself formatting

Programmers usually don't have to write data directly to or read data directly from a diskette using the BIOS. Generally, your application programs will be working with files. For this purpose, it's better to use the DOS functions. However, when you're formatting diskettes you must call various BIOS functions.

The DFP.PAS and DFC.C programs perform this task. These programs are replacements for the DOS FORMAT program. Similar to FORMAT, these programs not only format the diskette, but also create the various data structures that DOS expects. Among these are the boot sector, the root directory of the diskette, which is initially empty, and the FAT. For more information about these data structures and the general structure of mass storage systems under DOS, refer to Chapter 28.

DFP.PAS and DFC.C can process all known DOS formats (360/1200 on 5.25-inch diskettes and 720/1440 on 3.5-inch diskettes). Use the following model to call them:

```

      DFP Drive   Format   [ NV ]
      |         |         |
A: or B: |         |         |
      |         |         |
      360, 720, 1200, 1440 |         |
                        |         |
                        NV = No Verify |

```

Since both programs are based on the same algorithm and work with the same data types and constants, we'll discuss them together. Within the main program, the first argument of the DOS command line is analyzed. It's assumed that this argument will be the drive identifier (letter). This value is converted into the drive number (0 or 1). Next, the format of the drive is determined by using the GetDriveType procedure. To do this, we use function 08H of the BIOS disk interrupt, which returns a type code between 0 and 4. Within the program, this code is represented by the respective constants NO_DRIVE, DD_525, DD_35, and HD_35.

The older model PCs and XTs don't support function 08H. In this case, the carry flag is set after the function call to indicate an error. The program then defaults to a DD 5.25-inch drive that supports only a 360K format.

After GetDriveType() confirms the specified drive exists, the program then determines the logical and physical formatting parameters through the GetFormatParameter function. The format specified in the command line is passed to GetFormatParameter as a string along with the type code and two variables of type PhysDataType and LogDataType. You can see the organization of the two data structures in the Pascal version:

```

type DdptType = array[ 0..10 ] of byte;           { Structure for DDPT }
DdptPtr = ^DdptType;                             { Pointer to DDPT }

PhysDataType = record
    Seiten,                                     { physical format parameters }
    Spuren,                                   { desired side number of diskette }
    sektoren : byte;                          { Number of tracks per side }
    DDPT      : DdptPtr;                      { Number of sectors per track }
end;                                           { Pointer to Disk Drive Parameter Table }

```

You'll find the following program(s) on the companion CD-ROM



DDPTP.PAS (Pascal listing)
DDPTC.C (C listing)

```

LogDataType = record
    Media,
    Cluster,
    FAT,
    RootSize : byte;
end;

SpurBufType = array[ 1..18, 1..512 ] of byte;

```

{ DOS format parameters }
 { Media-Byte }
 { Number of sectors per cluster }
 { Number of sectors for the FAT }
 { Entries in the root directory }

PhysDataType contains the physical parameters needed for formatting. These are the number of sides, the tracks per side, and the number of sectors per track. Also, a pointer to the DDPT is stored here because both programs work with their own "private" DDPTs to speed up the formatting.

While the information in PhysDataType is needed for physical formatting, the information in LogDataType is needed for logical formatting, which applies to using different DOS data structures. That's why the DOS media ID, the number of sectors per cluster, the size of the FAT in sectors, and the number of entries in the root directory are recorded here.

The variables of type PhysDataType and LogDataType within GetFormatParameter are initialized with a series of typed constants (or STATIC variables in C), in which the necessary information for all supported formats is recorded. These constants, which are called DDPT_360, LOG_1200, or PHYS_720, can be identified quickly when looking through the listings. Before the procedure copies the parameters to the passed variables, it checks to determine whether the format specified in the command line is a valid one for the drive. The procedure displays the results of this test to the caller as its return value, which is FALSE if drive and format aren't valid.

After these checks, program execution continues by calling DiskPrepare. This procedure uses BIOS function 18H. Function 18H returns a pointer to the DDPT to the caller. The DDPT is part of the selected format. This pointer is ignored by the two programs, however, because a different DDPT is used by FormatGetParameter(). After DiskPrepare, the address of this DDPT is stored in the interrupt vector for interrupt 1EH. Remember the contents of this vector are first saved so the original DDPT address can be restored later.

Formatting then occurs using the PhysicalFormat function. The third parameter of the DOS command line indicates whether to verify the tracks. When formatting, PhysicalFormat uses the FormatTrack procedure to format one track at a time. FormatTrack calls the BIOS disk function 05H in a nested loop for tracks 0 to N; it formats side 0 first and then side 1 of each track. Obviously, it's also possible to format the entire first side and then the entire second side. However, this would take much longer because the read/write head would have to travel over the entire diskette twice. With this method, the drive is constantly switching between head 0 and head 1, but the formatting is performed much faster than moving the read/write arm from track to track.

After FormatTrack, PhysicalFormat calls the VerifyTrack procedure to determine the validity of the data (i.e., whether the contents of a sector and the corresponding CRC checksum match). This is performed only if FALSE is returned for the VERIFY parameter. VerifyTrack, like FormatTrack and WriteTrack, which we haven't discussed yet, is simply a procedure name for the corresponding BIOS function. If the BIOS reports an error, this call is repeated several times before a "real" error indication is finally returned to the caller. The maximum number of attempts is determined by the MaxVersuch constant, which is defined at the beginning of the listings. By setting this constant to one, both versions of the program will frequently report an unsuccessful format because errors occur more often than you might think.

Now let's return to the main program. If the diskette was perfectly formatted with PhysicalFormat, then the last processing step begins. This step involves logically formatting the drive using LogicalFormat. As we mentioned, in this step the different data structures that DOS needs for managing files are written to the diskette. We discuss this in detail in Chapter 28.

One interesting structure is the boot sector, which must be written to the diskette if it is to contain the DOS operating system. The contents of the boot sector are defined at the beginning of the program in the BootMaske variable. You'll find the details about the data and the small machine language program in Chapter 28. However, you should know the meaning of the

BootMes variable, which immediately follows the boot sector. This variable contains the string that appears on the screen when the computer is booted. It's written to the diskette with the boot sector.

You can alter the contents of this string. For example, add your own name or the name of your company so it appears on the screen when you boot the computer from the diskette. However, remember the end of the sector is indicated by a byte with the value 00H.

You'll find the following program(s) on the companion CD-ROM



DFP.PAS (Pascal listing)
DFC.C (C listing)

The end of LogicalFormat is essentially the end of program execution, because nothing happens in the main program except writing the status message.

Using BIOS To Access The Hard Drives

In this section we'll describe the BIOS functions for accessing hard drives. However, before we begin, we must warn you about experimenting with these functions. Unlike a floppy disk drive, in which you can insert an unused diskette for testing, a hard drive cannot be tested in this way. Using write and format functions carelessly can lead to irreparable data loss. Because of the structure DOS imposes on a hard drive, destroying one sector can cause all files and directories to disappear because DOS may no longer know where they are on the hard disk.

So, if you would like to "test" the BIOS functions, be sure to make a complete backup of your entire hard drive beforehand, or use another computer, if available. This is the only way you can avoid data loss, because even the most elaborate hard drive utility may not be able to help you.

The BIOS hard drive interrupt

As we mentioned, the hard drive shares interrupt 13H with the floppy disk drives. Although the functions for the hard drive and the floppy disk drives are identical, the BIOS controls the hard drive differently than the floppy disk drive. For this reason, the BIOS contains a module for controlling the hard drive and a separate one for controlling the floppy disk drives.

When interrupt 13H is called, the device number in the DL register determines whether a floppy or hard drive is being addressed. A value of 80H represents the first hard drive, while 81H represents the second hard drive. It's not possible to address more than two hard drives via the BIOS.

The functions of the hard drive BIOS have existed since the introduction of the XT. The original PC BIOS didn't have them. In 1981 no one thought of putting hard drives in microcomputers. When the AT and PS/2 model from IBM was introduced, some additional functions were added, as the following table shows:

Function	Task	Origin	Function	Task	Origin
00H	Reset	XT	0CH	Move read/write head	XT
01H	Read status	XT	0DH	Reset	XT
02H	Read	XT	0EH	Controller read test	only PS/2
03H	Write	XT	0FH	Controller write test	only PS/2
04H	Verify	XT	10H	Drive ready?	XT
05H	Format	XT	11H	Recalibrate drive	XT
08H	Check format	XT	12H	Controller RAM test	only PS/2
09H	Adapt to foreign drives	XT	13H	Drive test	only PS/2
0AH	Extended read	XT	14H	Controller diagnostic	XT
0BH	Extended write	XT	15H	Determine drive type	AT

Normal application programs don't usually access the hard drive through the BIOS. We'll describe only the most important functions in this section. You can find more information in the Appendix (located on the companion CD-ROM in which all the functions are described).

Status code

The hard drive functions use the carry flag to indicate an error. If the carry flag is set, then an error has occurred and the error status code is returned in the AH register. The following table lists the meanings of these codes:

Error codes when calling BIOS Disk Interrupt 13h for accessing the hard drive			
Code	Meaning	Code	Meaning
00h	No error	10h	Read error
01h	Function number or drive not permitted	11h	Read error corrected by ECC
02h	Address not found	20h	Controller defect
04h	Addressed sector not found	40h	Search operation failed
05h	Error on controller reset	80h	Time out, unit not responding
07h	Error during controller initialization	AAh	Unit not ready
09h	DMA transmission error.Segment border exceeded.	CCh	Write error
0Ah	Defective sector		

When one of these error occurs (except for error 1), you should first reset the drive and retry the function. Usually, the operation will then be successful. If error 11H is returned after a read function, the data isn't necessarily invalid. Actually, this status code indicates that a read error was detected, but was able to be corrected using an ECC (Error Correction Code) algorithm. This procedure is similar to the CRC procedure used by floppy disk drives. The individual bytes of a sector are calculated through a complicated mathematical formula. The resulting sum is written to the sector on the hard disk as four additional bytes. If a read error is detected, it can usually be corrected by using the ECC.

Using the hard drive functions

The hard drive functions also use registers for passing parameters. The function number is passed in the AH register. When the hard drive number must be identified, its value is passed in the DL register. The value 80H always represents the first hard drive, and 81H represents the second hard drive. The number of the read/write head and the side (0 or 1) are passed in the DH register.

The CH register specifies the cylinder number. Since you can represent only 256 cylinders with this 8-bit register and the hard drive of an XT has more than 306 cylinders, this register alone cannot specify the cylinder number. For this reason, bits 6 and 7 of the CL register are "appended" to the value in the CH register to determine the cylinder number. They form bits 8 and 9 of the cylinder number, so a maximum of 1024 cylinders (numbered 0 to 1023) can be addressed. Bits 0 to 5 of the CL register specify the sector number (1 to 17 per cylinder). If more than one sector is being accessed at the same time, the AL register specifies the number of sectors. In read and write operations you must also specify the address of a buffer, from which the data is written or to which the data are transferred. In this case, the ES register indicates the segment address and the BX register indicates the offset address of the buffer.

Resetting the hard drive controller

One function that doesn't require all the parameters is function 00H, which, like function 0DH, resets the controller. For example, after an error occurs, this function is routinely performed before the next data access. The only parameter needed is the hard drive number that is passed in the DL register.

Determining the status of the hard drive

Using function 01H, you can determine the status of the hard drive. Again, the drive number whose status is being checked is passed in the DL register.

Reading hard drive sectors

Function 02H reads one or more sectors of the hard drive. On each call to this function, you can read a maximum of 128 sectors. Perhaps you're wondering why the maximum is 128 instead of 256 sectors. The hard disk controller uses DMA capability to transfer data between the computer's memory and the hard drive. However, the DMA components can transfer a maximum of 64K of data at one time. This is equivalent to 128 sectors ($64K = 128 \text{ sectors} * 512 \text{ bytes/sector}$). Another restriction is the DMA components can transfer data only within a single memory segment. So, the read/write buffer is usually aligned to the start of a memory segment. Remember the ES:BX register pair point to the buffer. In this case, the ES register will point to the start of this segment and the BX register will have a zero offset.

When you use function 02H to read more than one sector per call, the sectors are read in the following order: First, the sectors in the specified cylinder and side are read in ascending order (by sector number). When the end of the cylinder is reached, the first sector on the same cylinder, but on the next head, is read. Sectors on the next cylinder aren't read until after the last head in the same cylinder is reached and there are sectors remaining to be read.

Writing hard drive sectors

Function 03H is used to write one or more sectors to the hard drive. This function is similar to function 02H except the data is written from the buffer to the hard drive. For a description of this function, refer to the one above.

Verifying hard drive sectors

Function 04H verifies the sectors of a cylinder. However, the data on the hard drive is compared with the ECC value instead of the data in memory (which is why it isn't necessary to specify a buffer address in ES:BX). The number of sectors to be verified is specified in the AL register.

Formatting the hard drive cylinders

A hard drive must be formatted before it can be used. Function 05H performs this task. This function is similar to the function for formatting a floppy diskette. The address of a buffer is passed in the ES:BX register pair. This buffer must be 512 bytes in size, although only the first 34 bytes are used. The buffer consists of two one-byte entries for each of the 17 sectors to be formatted. The first byte indicates whether the sector is good or bad. Before calling this function, we assume that each sector is good. So we store a zero value here. The second byte is the logical sector number.

Bytes 1 and 2 of the table are used when the first physical sector of the cylinder is formatted. Bytes 3 and 4 are used when the second physical sector is formatted, etc. So, while the physical sequence is fixed, the logical sequence of sectors is defined by the two bytes of a sector specification in this table.

The most obvious way to format the hard drive is to assign a sector's physical sector number to each logical sector. However, a technique called sector interleaving is actually used to speed up hard disk performance. We'll discuss sector interleaving in more detail in the "Hard Drive Advancements" section later in this chapter.

The first byte of each table entry may contain the value 00H, which indicates the sector is good, or 80H, which indicates the sector is bad. During formatting, this byte is transferred to the sector marker that indicates that DOS shouldn't use this sector to store data.

Determining the hard drive parameters

Unlike floppy diskettes, hard drives don't have uniform characteristics. For some programs, it's important to know the hard drive's parameters. To do this, use function 08H to pass the hard drive number in the DL register.

After calling the function, the DL register contains the number of hard drives connected to the controller. The value returned may be 0, 1, or 2. The DH register contains the number of read/write heads. Since this value is relative to 0, a value of 7 means there are 8 heads. The number of cylinders is returned both in the CL register (bits 0-7) and the two upper bits of the CH register (bits 8 and 9). Again, this value is relative to 0. Finally, the number of sectors per track is returned in the lower 6 bits of the CH register. This specifies the number of sectors per track, but is relative to 1, not 0.

Initializing a "foreign" hard drive

The BIOS in each computer already contains the specifications for various hard drives. This makes it easy to select the hard drive specifications during SETUP. However, suppose the specifications for a particular hard drive aren't in the BIOS. There's another way to make the drive's specifications known to the BIOS. First a table containing the specifications is constructed. Then the address of the table is stored at interrupt 41H or interrupt 46H, depending on whether hard drive 0 or hard drive 1 is being initialized. The format of the table is predefined by BIOS and describes the characteristics of the hard drive.

Finally, function 09H, which initializes the controller with the new hard drive specifications, is called. The drive number (80H or 81H) is passed in the DL register. Usually the device driver provided by the hard drive manufacturer manages this function.

Extended hard drive sector read/write

Functions 0AH and 0BH are similar to the read and write functions 02H and 03H. However, one difference is that, in addition to the 512 bytes of data per sector that's transferred, the four ECC bytes at the end of each sector are also transferred. Since each sector is 516 bytes instead of 512 bytes, the maximum number of sectors that can be read or written at a time is 127 sectors, while functions 02H and 03H can handle 128 sectors.

Function 10H tests whether the hard drive, whose number is passed in the DL register, is ready to execute commands. If the carry flag is set to indicate the drive isn't ready, then the AH register will contain the error code.

Recalibrating the hard drive

Function 0BH is used to recalibrate the hard drive. After the function call, this function returns the error status along with the drive number in the DL register.

Self test of the hard drive controller

Function 14H is used to perform a self test. If the controller passes the test, the carry flag will be reset.

The final hard drive interrupt function is 15H, which is available only on ATs, not XTs. This interrupt returns the drive type. The drive number (80H or 81H) is passed in the DL register. If the drive isn't available, a value of 0 is returned in the AH register. A value of 1 or 2 indicates a floppy disk drive. A value of 3 indicates a hard drive. In this case, registers CX and DX contain the number of sectors on this hard drive. The two registers form a 32 bit number, with the CX register containing the high-order byte and the DX register containing the low-order byte.

Hard Drives And Their Controllers

The advancements in hard drive technology are related to four types of controllers: ST506, ESDI, SCSI, and IDE. The format, in which the data are saved on the hard drive, not only depends on the controller, but also on the data transfer rate between the computer and the hard drive.

The following table shows the maximum data transfer rates that are possible with the different controllers. However, these are theoretical maximum values that not only are seldom attained, but also are affected by other factors. Imagine a set of data on its way from the hard drive to be displayed on the screen (e.g., when you load a document into a word processor). In addition to the controller, the program must interface with many levels: the BIOS, the DOS, the application program, and perhaps one or more TSRs.

Maximum data transfer rates of the various PC hard drive controllers			
Controller	Maximum Data Transfer Rate	Controller	Maximum Data Transfer Rate
ST506	1 Meg per second	IDE	4 Meg per second
ESDI	2.5 Meg per second	SCSI	5 Meg per second

Another factor that affects the data transfer rate is the speed of the bus. This limits the speed of hard drive controllers because at least the ISA bus still operates at 8 MHz, although the speed of the CPU has already reached the 100 MHz limit. So any published values quickly diminish to about a fifth by the time the data actually appears. Next, we'll introduce you to hard drive controllers, examine their structure and describe their advantages and disadvantages. The role of the BIOS is also discussed.

ST506 controller

The first hard drives that were widely accepted in the PC world were developed for the ST506 controller and compatible controllers. As its name indicates, this controller was manufactured from Seagate.

Even today, ST506 controllers are still the most widely used controllers, even though new hard drives usually are equipped with IDE controllers. Generally, hard drives designed for hookup to an ST506 controller are identified by the label "MFM/RLL". These letters refer to the two recording methods by which the controllers save data on the hard drive. Usually you can use DIP switches to set the format the controller uses. The RLL format is preferred because it provides higher hard drive capacity. (Refer to the "Recording Information On The Hard Drive" section in this chapter for more information.)

Because of its wide distribution, the ST506 controller has set different standards in hardware control systems, partly because the BIOS conforms to this controller type. The effects of this are evident today. For example, IDE and ESDI controllers are (and must be) compatible with ST506 controllers in various ways. We'll discuss this in more detail later.

The hardware

In the ST506 standard, the hard drive and controller are two separate components. The controller is on a separate card and occupies one of the PC's expansion slots.

A controller can usually manage two hard drives. Two types of cables connect the controller to each hard drive. Each hard drive connects to the controller with its own 20-pin data cable. If both hard drives are installed, they share the 34-pin control cable. The control cable sends electrical signals to the hard drive to select the desired read/write heads, search for the desired cylinder, etc. Data to be read from or written to the hard drive is transferred over the data cable in serial and analog mode.

The controller can convert the digital information on a cylinder into bit strings. The information on magnetic media exists as values of 0 or 1. The controller can reverse the digital values as needed; this process is called flux reversal.

The data transfer rate can be as high as 5 megabits/second using MFM recording and 7.5 megabits/second using RLL recording. Since the control information still must be "filtered out" from the stream of data, the effective transfer rate of useful data is considerably lower. Even so, MFM controllers can process .5 megabytes per second and RLL controller can even process 0.75 megabytes per second. However, usually these theoretical values ignore factors such as head select time, cylinder seek time, etc., and assume that you'll read contiguous sectors. We'll discuss this in more detail in the section on interleaving.

The higher transfer rate of RLL controllers is a result of a more efficient recording scheme. You'll see that more sectors can be written to each track using RLL. MFM drives can write 17 sectors on a track, but RLL drives can fit 26 sectors on each track. In both formats, the rotational speed of the drive is 3600 RPM.

When the XT first appeared, the only controller available was the ST506. As a result, the new hard drive functions of the ROM-BIOS were developed specifically for this controller. These hard drive functions have imposed rigid limitations on PC manufacturers. For example, the number of drives is limited to two, the maximum number of cylinders to 1204, the maximum number of sectors per track to 63, and the maximum number of heads to 16. Also, the sector size is fixed at 512 bytes. When combined with all the other factors, the maximum capacity of a drive is 504 Meg.

To overcome these limitations, some hard drive controllers "trick" the system into believing there are two hard drives that are on the same drive. This makes it possible to have a hard drive with a capacity of up to 1 Gigabyte. However, an ST506 controller is too slow for a drive with such enormous capacity. So, it isn't practical to connect such large hard drives to systems with ST506-type controllers. Because of this, the ST506-type controller will gradually disappear in the coming years.

ESDI controllers

The ESDI controller was developed after the ST506 controller. ESDI, which is an acronym for "Enhanced Small Devices Interface", is found in many IBM PS/2 models. This controller represents an advancement of the ST506 model. Generally, ESDI controllers are compatible with the ST506 and can be used in computers whose BIOS is programmed to support only ST506 controllers.

Unlike the ST506, ESDI doesn't transfer every flux reversal serially to the controller via the data line. Instead, part of the decoding logic, called the data separator, is already on the hard drive. The data separator prepares the data read from the hard drive and transfers only the useful data, in digital form, to the controller.

Because the controller and the data separator work in parallel, the transfer rate can be as high as 10 megabits/second. This represents only a doubling of the transfer rate of an MFM ST506 drive. However, this is accompanied by another performance enhancement. ESDI controllers usually have a sector buffer that makes an interleave factor of 1:1 possible. What does this mean? An ST506 drive with an interleave factor of six requires six full revolutions to transfer the contents of an entire track. With an interleave factor of three, this same drive still requires three full revolutions to transfer the contents of an entire track. However, an ESDI can perform the same task in a single revolution. This results in increasing the access speed by a factor of three to six.

Also, some ESDI systems can reach a transfer rate of 15, 20, and even 24 megabits per second. However, such controllers are rare and usually quite expensive. So most ESDI controllers work with 10 megabits.

The data separator isn't the only intelligent component on an ESDI system. An ESDI hard drive also stores information about its physical format and the addresses of defective sectors and can send this information to the controller. Then the controller can perform its own SETUP, which is a task that a user has to do with ST506 drives.

The BIOS and ESDI controllers

The hard drive information is then stored in the CMOS RAM of an AT. The BIOS must know the parameters of the hard drive and pass this information to the DOS device driver.

Because the ESDI controller can request this information from the hard drive, the information in the BIOS doesn't have to match the actual characteristics of the drive. Often this discrepancy cannot be avoided, because each BIOS knows only a limited number of hard drive types and their specifications.

You'll encounter problems on an ST506-type hard drive if the installed drive doesn't appear in the BIOS list. In this case, you must select a drive, from the list, whose specifications most closely match those of the installed drive. This often means "wasting" some of the drive capacity. You must select a BIOS entry for a drive with fewer cylinders, tracks, or heads. Otherwise the system might try to access sectors that don't even exist on the installed drive, which results in an error.

Another problem is when the BIOS doesn't have a hard drive type that works with the same number of sectors per track. Then you must select a BIOS entry for a drive with the next smallest sector size, which means wasting valuable sectors in every track on the drive and increasing the number of unused sectors to an undesirable level.

With ST506 drives, this isn't a problem because they only work with 17 or 26 sectors, depending on the recording method. However, ESDI drives have 34 or 36 sectors per track. So, if the BIOS contains entries for drives with only 26 sectors, then you might end up wasting one-third of your expensive hard drive capacity.

Fortunately, most ESDI controllers can avoid this problem. The BIOS entry for a drive with a slightly smaller capacity than the ESDI hard drive is selected during SETUP. Since ESDI hard drives can send their specifications to the ESDI controller, the controller knows the physical characteristics of the drive. Using a special feature called sector translation, the ESDI controller converts the BIOS logical drive specifications into the hard drive physical specifications. The conversion takes slightly longer, but ensures the ESDI hard drive can be used with almost any BIOS entry without wasting a lot of its capacity.

The ability to perform this conversion depends entirely on the ESDI controller. Some controllers support only certain BIOS entries, while others are more flexible and can accept any format. This is also an advantage if an ESDI drive encounters the

limits imposed by the BIOS hard drive functions, for example, an ESDI drive with more than 1024 cylinders. Instead of wasting the cylinders above 1024, the controller simulates a greater number of sectors per track. Although the ESDI drive may have only 34 tracks per sector, it can pretend to have up to 63 tracks per sector. The ESDI controller then translates a logical cylinder/sector/head specification to the hard drive's physical specification.

SCSI controllers

The SCSI controller (pronounced "scuzzy") standard, isn't really a hard drive interface. Instead, it's a way to connect up to eight entirely different devices to a PC. Besides hard drives, you can connect tape backup streamers, CD-ROM drives, or scanners to a SCSI interface.

Unlike other hard drive controller standards, the SCSI (Small Computer System Interface) isn't found only in PCs, but also on many 68000 systems (Macintosh and Atari ST) and large workstations. One reason for its appeal is that you can easily couple and uncouple devices from the SCSI interface because the devices communicate with the controller through a bus that's separate from the PC bus.

Both the SCSI line specifications and the SCSI commands to control the devices are standardized. SCSI devices can be easily exchanged between different systems; only the SCSI controller must be matched to the host computer system.

The SCSI bus, which links different SCSI devices together, is usually a cable with an 80-pin plug. The bus allows 8-bit parallel data transfer. A new version, called SCSI2 allows 16-bit parallel data transfer.

Manufacturers of SCSI controllers like to tempt their customers by quoting data transfer rates of 4 or 5 Meg per second. Actually, these rates aren't possible. While the SCSI controller can handle these high data transfer rates, the hard drive connected to it cannot. So, you can expect a data transfer rate of between 1.5 and 2 Meg per second, unless you purchase an expensive EISA controller, which can manage 2.5 Meg per second.

SCSI drives continue the trend started by ESDI drives by integrating much of the control circuitry directly on the hard drive. Actually, this must be done with this system, because the controller must remain device-independent and not concern itself with the specific characteristics and features of a hard drive.

Shorter paths aren't the only advantage provided by the close proximity of the hard drive and its control circuitry. This also makes it easier to trick the controller into using a certain disk format that doesn't really exist. Like an ESDI controller, the SCSI controller asks the attached devices for its specification when the system starts and passes this information to the requester.

SCSI systems depend on their own built in BIOS. From a software standpoint, SCSI systems don't support the ST506 standard of the ROM-BIOS. The original functions of the BIOS Disk Interrupt are replaced by the SCSI BIOS. Unfortunately, the SCSI BIOS isn't useful in protected mode, in which "operating environments" such as Novell or OS/2 access the disk directly and often support only the ST506 standard. This requires a specific device driver, which may not be available for the particular operating environment. This is one of the biggest disadvantages of the SCSI interface.

However, if you have the required driver, the hard drive specifications are automatic. To install a new SCSI drive, simply connect it to the bus. The SCSI controller handles the rest by using the correct "driver ware".

IDE

The new star of hard drive controllers is the IDE controller (Intelligent Drive Electronics) interface. The IDE interface is found in almost every new PC. Its development started in 1984, when PC manufacturer Compaq asked Western Digital to develop an ST506-compatible controller that would fit on the hard drive to save space.

Using an IDE drive provides a hard drive and a controller in one. A single 40-pin cable combines the functions of a data cable and a control cable and connects the IDE drive directly to the system bus.

This is also where IDE drives get their nickname; sometimes they're called "AT Bus Drives". But IDE drives aren't limited to ATs with their 16 bit data buses. You can also use IDE drives on an 8-bit XT bus.

Many PCs have a connector for an IDE cable directly on the motherboard. With other PC's you must use a small expansion slot board, which then connects to the IDE cable.

Combining the drive and controller gives the IDE some of the same advantages of the SCSI controllers. Among these are the ability to emulate any drive format and track caching, in which the drive reads an entire track and keeps the data in an internal cache buffer until it's needed. IDE drives can operate with an interleave factor of 1:1, which provides faster access. IDE combines the advantages of the other three standards; it's flexible like SCSI, fast like ESDI, and is compatible to the ST506 standard so it can be connected to most PC systems easily.

Also, IDE drives have a very low power consumption, so they're ideal for laptops and notebooks.

Many IDE drives have special commands for working with laptops and notebooks. For example, these commands can put the notebook computer "to sleep", which minimizes power consumption. These commands are generally used in connection with special drivers or a BIOS that's adapted to work with IDE drives. From the BIOS' point of view, IDE drives behave like normal ST506 controllers, making it easy to integrate them into existing systems.

New standards are being defined for IDE drives. In the near future we'll most likely see new BIOS systems supporting IDE drives directly. Then PC makers can fully use the extended features offered by these drives, which go unused in many of today's systems.

From controller to memory

Regardless of the speed of the hard drive and the controller, the way in which the data is transferred by the controller to the memory determines the effective speed of a controller-hard drive combination. Four different methods can be used to do this:

- | | |
|------------------------|---------------------|
| ➤ Programmed I/O (PIO) | ➤ Memory Mapped I/O |
| ➤ DMA | ➤ Busmaster DMA |

Programmed I/O

With programmed I/O, the different controller I/O ports manage both the drive commands and the transfer of the data between the controller and the main memory. If you use programmed I/O, you'll use the IN and OUT assembly language instructions. This means that every byte or word must be channeled through the CPU.

Here, the data transfer rate is limited by the speed of the PC bus and the performance of the CPU. While the ISA bus allows a maximum transfer rate of 5.33 Meg per second (16-bit rate), this rate is unattainable with any of today's CPUs. With fast 386es, 486es or Pentiums, the data transfer rate is limited to about 3 or 4 Meg and can be attained only with very fast and expensive hard drives.

Memory Mapped I/O

The CPU can process data from a disk controller even faster if it stores them in a fixed memory region. The segment located above the video RAM is generally used for this purpose. Data in a program's memory area can be transferred faster using MOV instructions. This is faster than accessing the I/O ports with IN and OUT.

Even by using memory mapped I/O, today's speedy CPUs can request data faster than the controller is capable of transferring. The controller can never reach the theoretical maximum value of 8 Meg per second. It can't even reach 5 to 6 Meg per second.

DMA

DMA (Direct Memory Access) transfer is more widely known than the first two methods. Using DMA, a device (hard drive, floppy disk drive, CD-ROM, etc.) can transfer data directly to the computer's memory. The CPU is bypassed. To use DMA, a program only needs to tell the DMA controller how many bytes should be transferred from one location to another. This makes DMA seem like the best method for transferring data.

However, the DMA controller in the PC is inflexible and slow. In fact, it's so slow that Programmed I/O is faster on 386 and higher systems. The DMA controller operates at 4 MHz on the AT or later systems even though it worked at 4.77 MHz on earlier PCs. So, DMA transfer is faster than Programmed I/O only on PCs. Using DMA, the data transfer rate is limited to about 2 Meg per second.

Therefore, the DMA method is no longer used on most modern hard drives, even though many hard drives do support this method in addition to Programmed I/O.

Busmaster DMA

Busmaster DMA is another form of direct memory access, but isn't related to the DMA circuitry on the motherboard of the computer. Using this method, the hard drive controller disconnects the CPU from the bus and transfers data to memory on its own using its own Busmaster DMA controller. Transfer rates of up to 8 Meg per second are possible. Unfortunately, this feature increases the price of the controller. Busmaster DMA is generally used only with very powerful SCSI controllers.

Protected mode

The methods of DMA transfer are limited to real mode programs. Without special intervention, DMA cannot be used in protected or virtual mode. How virtual memory management is performed by the CPU's memory management unit (MMU) is responsible for this limitation. The MMU maps a program's virtual memory addresses into real physical memory addresses.

A protected or virtual mode program doesn't realize this, because it's never aware of the physical addresses; it works only with virtual addresses. When this program wants to perform a DMA transfer, it passes a virtual address to the DMA chip. However, because the MMU isn't involved in the DMA transfer, it cannot map this virtual address into a physical address. As a result, the data is transferred to a different memory area. The system will soon crash because important memory areas are overwritten.

This is a characteristic of protected mode operating systems, such as Windows and OS/2. However, this also occurs in Virtual 86 mode using DOS if the EMM386.EXE device driver is used to emulate expanded memory. EMM386.EXE depends on the virtual memory management of the processor.

The solution is to "watch" the DMA controller. Do this in protected mode to control all the I/O ports. In Windows, for example, a virtual control monitor in the background is installed to watch the programming of the DMA controller via the BIOS or another program. This monitor converts the actual physical addresses before they are written to the register of the DMA controller.

Recording Information On The Hard Drive

To understand how information is written on a hard drive, you must first forget the concept of binary coding. Zeros and ones aren't stored on the magnetic surface of a hard drive. It's impossible to represent these two states as "magnetized" and "not magnetized".

Why isn't this possible? If you try to represent data as sequences of magnetized and non-magnetized particles, the read head of the hard drive wouldn't be able to keep the individual magnetic particles separate. So, it wouldn't be able to distinguish between three or five zeros.

One way to avoid this problem is by knowing the length of a magnetic particle and the elapsed time for each magnetic signal. In other words, you need a kind of clock that indicates, with each tick, that it's time for a new bit.

However, the constant period of time for a cycle cannot be clearly defined because of various factors. For example, the rotational speed of the hard drive may vary slightly. But a bigger factor is that a single magnetic particle can never be magnetized; only a group of magnetic particles, whose number isn't always constant, can be magnetized.

It is possible, however, to record flux reversals, which are short passages between non-magnetized particles and magnetized particles. The reversals in flux create an electrical pulse in the hard drive's read head. This pulse is then passed to the electronic circuitry, where it is used to decode the stored information as zeros and ones.

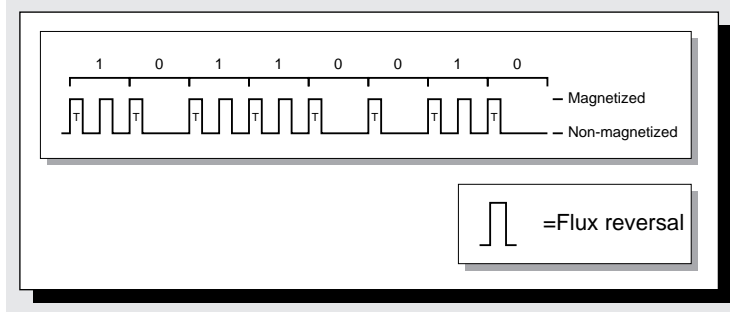
This coding and decoding of binary information has always been important to hardware developers. The number of flux reversals that can be recorded per square inch on a hard drive is limited. This limit depends on the composition of the magnetic material, the gap of the read/write head, its sensitivity, etc. Anyone who can find ways to record more zeros and ones on the hard drive with an equal number of flux reversals will lead the competition in higher and higher disk capacities.

FM method

The simplest way to encode zeros and ones to a magnetic surface is to record a flux reversal for each one-bit and omit a flux reversal for each zero-bit. However, you'll encounter a problem when you want to record a long series of zeros. In this instance, you must omit many flux reversals to represent the string of zeros. This would confuse the controller that depends on flux reversals to keep in sync with the data on the hard drive.

To avoid this problem, a clock signal is written onto the drive along with the data. Using the FM method (Frequency Modulation) recording technique, a one-bit might be recorded as two consecutive flux reversals and a zero-bit recorded as a flux reversal followed by no flux reversal. In both cases, the initial flux reversal represents the timing signal. The data bit is then modulated "between" the timing signal (second flux reversal for one-bit and no flux reversal for zero-bit).

The FM method of recording



Although this method is simple and inexpensive, it has one major disadvantage. Each data bit requires two flux reversals, which reduces the potential disk capacity by half.

MFM method

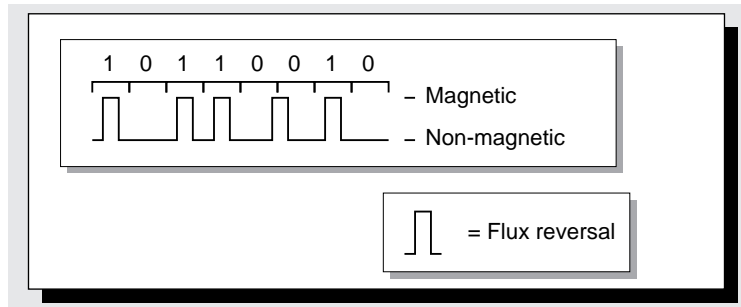
To reduce the number of flux reversals of the FM method and thereby increase the density at which information can be recorded, another encoding technique is used. This technique is called MFM method (Modified FM). Basically, the data is encoded as follows:

Code table for the MFM	
Data bit value	Encoded as:
1	Flux reversal
0 following another 0-bit	Flux reversal followed by no flux reversal
0 following another 1-bit	No flux reversal followed by no flux reversal

With this method, the timing signal is also used to store data. Both zero-bits and one-bits are recorded using only a single flux reversal. Longer sequences of zeros and ones appear as a continuous sequence of flux reversals. This encoding method requires improved control circuitry so the hard drive can synchronize with the normal sequence of timing flux reversals for longer sequences of zeros. The only problem is when a one is followed by a zero, which requires a flux reversal to the normal timing position. The time between the flux reversal of the one and that of the zero amounts to only half the normal interval between two flux reversals. However, this won't work because the shortest interval between two flux reversals cannot be shorter; otherwise the electronic circuitry would no longer be able to keep up. Therefore, a flux reversal isn't stored for a zero that follows a one.

The next flux reversal comes after one and a half times the time for a flux change (bit combination 100b) or even after twice the time (bit combination 101b). The following illustration demonstrates this.

The MFM method of recording



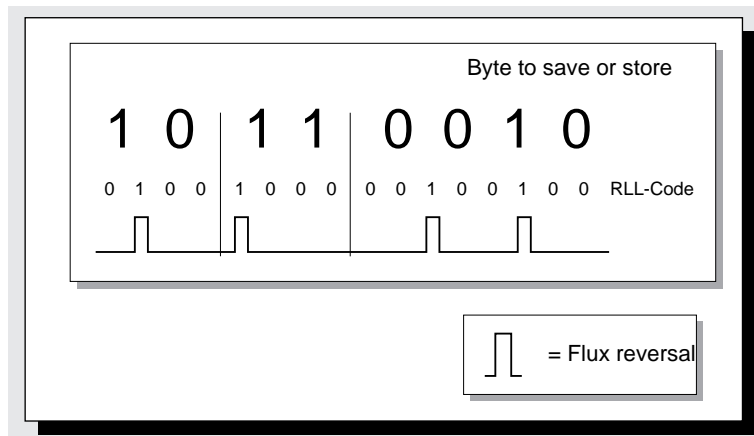
RLL method

Another encoding method, called RLL method (Run Length Limited), packs up to 50 percent more information on the disk than MFM.

In RLL, ones are stored as flux reversals; zeros are stored as the absence of flux reversals. A timing signal isn't recorded. The hard drive circuitry itself supplies the timing reference. However, even with a very constant rotation of the hard drive and improvements in the read head and its electronic circuitry, this is possible only if there aren't too many zeros between two ones. With each zero, the time to the next flux reversal increases, and along with it, the possibility the hard drive controller will lose its beat.

However, the ones cannot follow each other too rapidly. Otherwise, the controller may not be able to keep pace.

RLL encoding method of recording



Instead of encoding a single bit, the RLL method looks at and encodes a group of data bits. This group or bit string is from 2 to 4 bits in length. The new encoded string is twice as long as the original, but ensures the sequences of zeros aren't too long and the distances between ones don't become too short (and thereby overwork the circuitry). The 2,7 RLL encoding scheme is today's standard and is used in most modern hard drives. In the encoded string, a minimum of 2 and a maximum of 7 zeros will appear between two ones. This method increases the capacity of a drive by about 50 percent compared to MFM.

Another scheme is 3,9 RLL, which is called "advanced RLL", lets you fit even more information on the drive. With this method, the encoded scheme has a minimum of 3 and a maximum of 9 zeros between two ones.

The table on the right shows the encoding scheme for RLL 2.7. At first, you may think that a byte such as 00000001b cannot be encoded. However, don't forget that, on this level, you're working with sectors instead of single bytes. So, it's even possible to encode a byte, such as 00000001b, by including the bits of the following byte in the coding.

The only problem in coding occurs with the last byte of a sector because this method needs the following byte. This problem is solved by simply using a byte from the hard drive controller. The excess bits are simply truncated so the last byte of a sector is correctly decoded.

Code table for the RLL 2.7			
Bit pattern	Encoded as	Bit pattern	Encoded as
000	000100	11	1000
10	0100	011	001000
010	100100	001	00001000
0010	00100100		

Hard Drive Advancements

One of the remarkable achievements in PC computers has been the "smaller, faster, cheaper" advancements in hard drive technology. For instance:

- Access times in the top performance models have decreased from over thirty milliseconds to about ten milliseconds.
- Increases in maximum storage capacities. Hard drives with a capacity of more than 1 Gigabyte are now common.
- 1 Meg of hard drive capacity costs less than ten dollars.

The hard drive's performance and capabilities has increased because of the optimization in recording and playback techniques. In this section we'll discuss some of these techniques.

Interleaving and the interleave factor

Today hard drive controllers are so fast they can read an entire track even when only the data in a single sector is requested. So, when the data in the next sector is needed, the hard drive controller doesn't have to read the next sector. Instead, it can take the data from the controller's internal buffer. This significantly speeds up data access.

Most ST506 controllers have only an internal sector buffer (with a capacity to store the contents of exactly one sector), so only the newer controller types (SCSI, ESDI, and IDE) are track buffer compatible. Also, the sector buffer cannot be reused until the last byte of the previous sector is passed to the CPU. So, additional time is required until the next sector passes under the read head. The data in the subsequent sector cannot be read until the sector passes under the read head again; this requires almost an entire revolution of the hard drive. This process continues with each sector, noticeably reducing the speed of disk access. To minimize this delay, interleaving is used to spread the logical sectors across the track.

By interleaving the logical sectors, the controller has enough time to process and transfer the data in one sector before the next logical sector passes under the read head. This avoids the rotational delay of having to wait for a complete revolution before the sector can be read.

Interleaving is measured by the interleave factor. This value is the number of sectors by which the logical sector number was shifted, compared to the physical sector number. The original XT hard drive uses an interleave factor of 1:6, while the AT

Interleaving in the first XT and AT hard drives from IBM			
AT		XT	
Physical sectors	Logical sectors	Physical sectors	Logical sectors
1	1	1	1
2	7	2	4
3	13	3	7
4	2	4	10
5	8	5	13
6	14	6	16
7	3	7	2
8	9	8	5
9	15	9	8
10	4	10	11
11	10	11	14
12	16	12	17
13	5	13	3
14	11	14	6
15	17	15	9
16	6	16	12
17	12	17	15

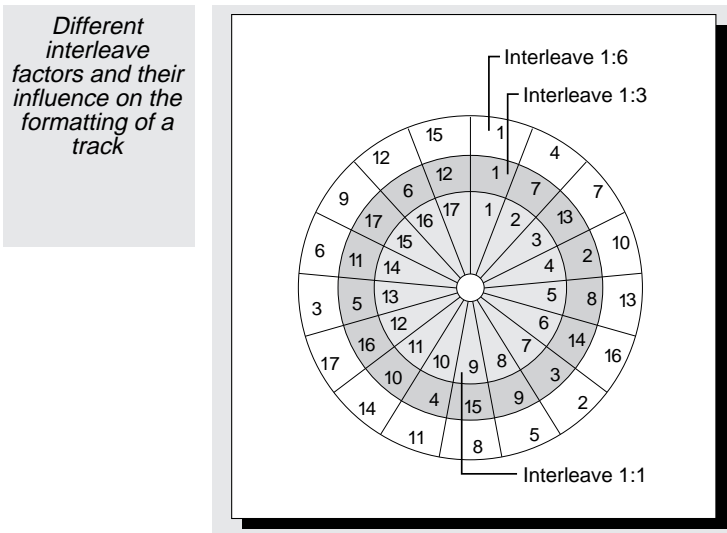
uses an interleave factor of 1:3. However, the interleave factor of the AT hard drive can be reduced to 1:2, which increases the access speed. Today, interleave factors of 1:1 are common, which means there is no interleaving at all.

However, an interleave of 1:6 requires that five physical sectors be skipped before the next logical sector can be read. An interleave of 1:3 requests that two physical sectors be skipped. The table on the previous page shows the logical arrangement of sectors on a track with 17 sectors.

Despite the advantages of this method, the best interleave is actually no interleave at all. Without an interleave, an entire track be read within a hard drive revolution. However, two, three, or even more revolutions would be needed with interleaving, depending on the interleave factor.

Setting the interleave

Set the interleave in the "low-level-format" of the hard drive, which creates the address labels and sector numbers on the surface on an unused hard drive.



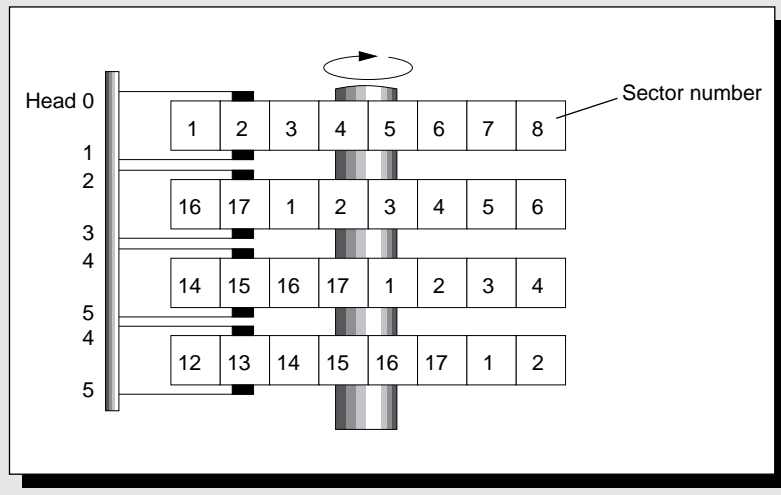
Because the logical sector number of a single sector is determined by the low-level-formatting program, it's possible to shift the sectors. Many hard disk utilities will prompt you for the desired interleave factor. If you select a "bad" value, your hard drive may slow down when loading programs or accessing files.

Even if you do select a "bad" interleave factor, you can still eliminate the problem later. You can use a program, such as the Norton Utilities, to do this. Such programs determine the optimum interleave factor and then execute the appropriate low-level-format for you. This usually takes just a few minutes.

You may think that this type of utility will lead to data loss because the program is "tampering" with the hard drive. However, as long as the program is error-free, you shouldn't lose any data. Since these programs work below DOS level, DOS is completely unaffected by the changes. Remember, DOS doesn't know the origin of a sector which it reads or writes. So, as long as the data are read before a track is reformatted and written back, DOS believes that nothing has changed.

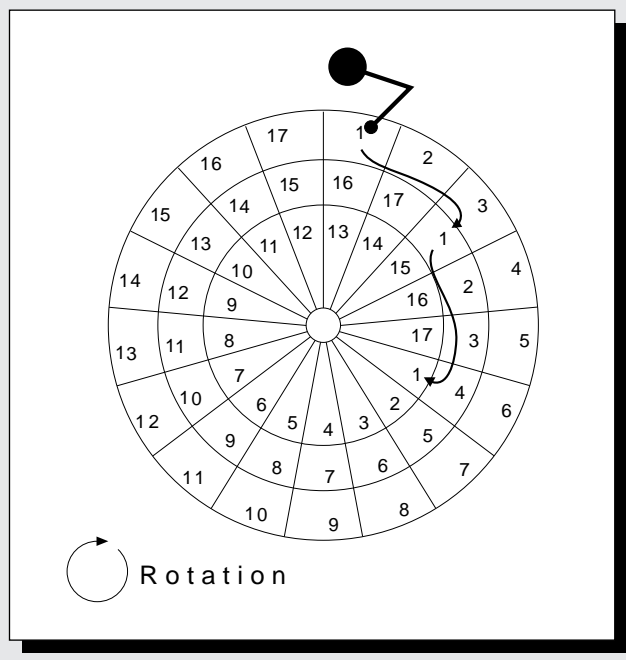
Track and cylinder skewing

Carefully selecting an interleaving factor significantly increases hard disk performance. After the data from one track is read, data, which is in same cylinder but the following head, are usually accessed. As the controller switches the read/write head, the hard drive continues to rotate. After reading the last sector on the track, the first sector on the next head has already passed the read/write head. So you must wait almost an entire revolution.

Cylinder skewing

To prevent this from happening, use cylinder skewing. All of the sectors on a track are shifted so, after switching to the next head, the first sector can be read without any rotational delay. You can set the cylinder skewing during the low-level-formatting of the drive.

In addition to cylinder skewing, there is also track skewing. This is similar to cylinder skewing, except that it considers the time needed by the drive to move the entire read/write arm to the next track.

Track skewing with an interleave of 1:1

Multiple zone recording

One way to increase the capacity of a hard drive is to format more sectors on the outer tracks. Since they have a larger circumference, the outer tracks have more space than the inner tracks. With ST506 controllers, the number of sectors per track was fixed and limited by the capacity of the innermost (shortest) track.

With modern SCSI and IDE disks, which only simulate the number of heads, tracks, and sectors, it's possible to format individual tracks with a different number of sectors. These controllers and drives translate a logical head, cylinder, and sector number to a physical head, cylinder, and sector number. So, it's possible to vary the number of sectors on outer tracks.

This places a greater demand on the controller circuitry and the read/write head. In the same (rotational) time, more sectors must be read or written to the outer tracks than on the inner track. So, while the length of the flux reversal decreases, the number of data bits to be read or written increases.

This can increase the capacity of a hard drive by between 20 and 50 percent compared to the usual, fixed sector per track arrangement.

Error correction

One important indicator of hard drive performance is its reliability. Impurities in the magnetic material are unavoidable and cause some sections of the hard drive to be unusable.

In the past, a list of defective sectors were printed on the hard drive case as they were detected by the manufacturer. During the low-level format, the user entered these sector numbers so the operating system would recognize these defective areas. Then the operating system would mark them as defective in the FAT (File Allocation Table) and avoid using them for storing data.

Newer hard drives, especially IDE models, no longer have a defective sector list. Either the sectors are already recorded on a separate data track by the manufacturer or they are recognized during the low-level format. These defective sectors are either skipped or replaced by other sectors from "alternate" areas. The tables that identify the defective sectors and their alternates are passed to the hard drive controller during initialization. When the hard drive controller tries to access a sector identified as defective, it can switch to the alternate sector. Although this slows access to such a sector, since there are very few defective sectors, this is hardly noticeable.

Many drives are also able to recognize defective sectors during read and write operations. This is the purpose of the ECC (Error Correction Code) that is automatically generated and saved with each sector. If an error is encountered during an operation, in many cases the data is reconstructed, an alternate sector is assigned, and the data is rewritten to this alternate sector. This feature is found in most of the IDE drives.

Other hard drive components

A hard drive contains more than just the data bit information. When a sector is formatted, an entire set of information is also written so the hard drive correctly recognizes and reads data. Although the format of this information is different depending on the controller, the information itself is the same (e.g., the cylinder number, the sector number, and the error correction code).

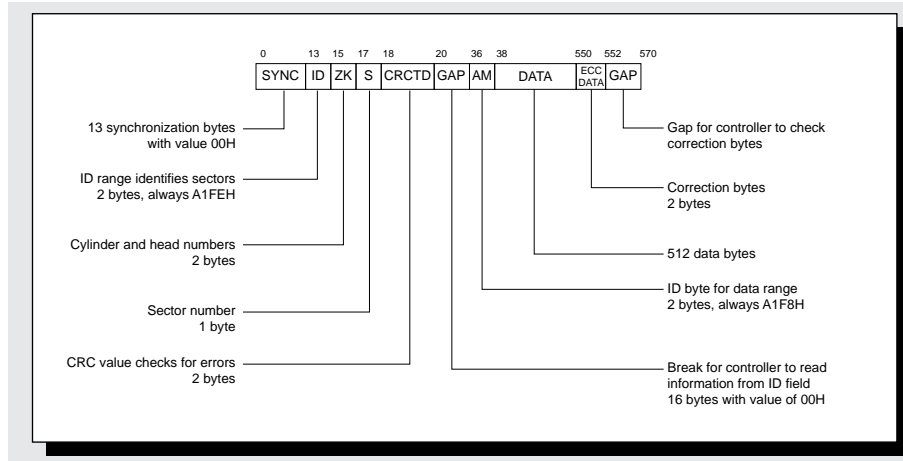
Each sector begins with a series of thirteen synchronization bytes with the value 00H (SYNC field). This bit pattern results in a constant series of flux reversals that help the controller synchronize itself to the sector.

Following this is the ID field, which identifies the sector. The cylinder, head, and sector numbers are recorded here (CH/S fields). This information starts with a special ID byte (ID field) and ends with a two byte error code that's used to check the validity of the information (CRCID field).

Next is a gap giving the controller a break before the start of the data field. The controller needs this break to read the information from the ID field to determine whether this is the desired sector. The gap also lets the controller resynchronize itself. The actual data field follows. The data field also has an ID byte (AM field) at the start, which is for identification. The 512 data bytes (DATA field) are next. The data field ends with two correction bytes (ECC-DATA field) the controller uses to determine whether the data is valid.

The sector ends with another gap to give the controller time to check the correction bytes. Here the gap contains bytes with the value 4EH. So the sector's total length is 570 bytes.

*Sector format
created by an
ST506-type
controller*



Extra tracks

A hard drive also contains other "reserved" areas. For example, there are servo tracks, which are used by the hard drive's electronic circuitry to synchronize itself to the data clock. This is especially important when used with RLL controllers and drives.

Some drives also contain "reserved" or "alternate" tracks. These tracks are unknown even to the BIOS. The area on a reserved track is used as a replacement for defective sectors.

Finally, there is the park area. When the computer is switched off, the read/write heads "land" or settle on the surface of the park area. If the read/write head lands on a track containing data, that data might be destroyed or damaged. The park area contains no data and is a safe area that cannot be scratched or otherwise damaged by the read/write heads.

Access times

A hard drive's performance is measured by its access speed. Hardware manufacturers often claim their hard drives have access speeds of 10 or 30 milliseconds. This value represents the average seek time between two file accesses.

Most manufacturers also use track-to-track seek time and maximum seek time to express hard drive speed. The track-to-track seek time is the time needed to move the read/write head from one cylinder to the next. The maximum seek time is the time needed to move the read/write arm from the first cylinder to the last cylinder of the drive.

Although these factors have an important effect on how a hard drive accesses a file at the DOS level, there are other important factors. For example, a hard drive with extremely fast specification is useless unless the controller can keep pace with it. Also, a hard drive could be so fragmented the read/write arm is constantly jumping back and forth between different cylinders to access data. So, performance must also be measured according to the time that a read or write request spends at the various levels of the application program, at the DOS level with its device drivers, the BIOS, and any special hard drive drivers up to and including the programming of the hard drive controller.

To verify the performance data of a hard drive manufacturer, you can use a performance measuring program, such as SystemInfo from Norton Utilities or CORETEST from Core International.

Generally these programs measure a hard drive's data transfer rate by reading the largest possible data block from the hard drive. However, the size of this block is restricted to one complete cylinder, which means the read/write arm doesn't move during operation. We already know that ESDI, SCSI, and IDE controllers often "hide" the true specifications from the BIOS and instead translate the drive's cylinder, head, and sector values into physical ones. So, the results of these tests may not

be accurate. A controller translation that results in a track change is also accompanied by a track-to-track delay, which distorts the result of the measurements.

When measuring the track-to-track seek time, you'll encounter a similar problem. To do this, read two sectors in adjacent tracks. However, the controller translation of these two sectors may not result in a track change, which produces an unrealistic zero track-to-track seek time.

Cache controllers and cache programs

The results of these measurements are especially suspicious when cache programs are used. At the software level, disk cache programs, such as SMARTDRV or PCKwik, can have a major effect on disk drive performance. These cache programs hook into the BIOS hard drive interrupt and intercept the read and write calls of the application programs and the device drivers of DOS.

When an application program wants to read data from a hard drive, the cache program intercepts the read request, passes the read request to the hard drive controller in the usual way, saves the data that was read in its cache buffer, and then passes the data back to the application program.

Depending on the size of the cache buffer, numerous sectors are read into and saved in the buffer. When the application wants to read more data, the cache program again intercepts the request and examines its buffers to see if the data is still in the cache. If it is, the data is immediately passed back to the application without another hard drive operation. As you can imagine, this speeds up access tremendously and can greatly affect the disk drive performance measurements.

In order to maintain accurate measurements, you can write a program to check the BIOS disk interrupt 13H. If it's still pointing to the ROM BIOS, then a cache program isn't active. If it's not, then you can request the user remove or disable the cache program until the measurements are completed.

Another type of disk cache is one that is part of the hard drive controller. This is a hardware disk cache and doesn't use any BIOS interrupts. Instead, the caching is performed at the hardware level and is invisible to normal performance measurement software.

Hard Drive Partitions

If you've ever installed a hard drive or added an operating system, such as XENIX or OS/2, then you've used the DOS FDISK command. This command is used to partition the hard drive. A hard drive must be partitioned when you want to logically divide the hard drive into separate volumes or when you want to install more than one operating system on the same hard drive.

Formatting process

To prepare a hard disk to be used by an operating system, you must perform three tasks. First you must perform a low-level format. When you do this, you are organizing the drive into cylinders, tracks, and sectors by writing the appropriate address markers onto the drive surface. The address markers are later used by the controller to identify the specific sectors.

In the early days of the PC, the DOS DEBUG command was used to perform a low-level format. Today, low-level formatting is no longer complicated because most hard drive manufacturers provide programs that perform this task.

Partitioning the hard drive

Next, you must partition the hard drive. There are two reasons why this is necessary. First, it enables you to install multiple operating systems, such as DOS and XENIX, on a single drive. Partitioning a hard drive into separate areas lets each of the operating systems manage its disk space without the conflicts caused by different file structures.

The other reason for partitioning a hard drive is to be able to use the additional capacity of larger drives. The original XT had a hard drive with a 10 Meg capacity. Then drives with 40 and 80 Meg capacities appeared. Early versions of DOS were able to manage hard drive capacities of only 32 Meg. But this limitation was addressed by DOS 3.3. Now the 32 Meg maximum capacity applied only to a partition. DOS 3.3 allows one primary partition with a 32 Meg maximum and an extended partition

that can be divided into as many as 23 logical devices (drives C: through Z:). Each of these logical devices can hold up to 32 Meg, which makes the entire hard drive capacity 768 Meg.

DOS 4.0 goes even farther, supporting drives with a maximum capacity up to 2 Gigabytes. Nevertheless, many users continue to partition the hard drive into logical drives, because they would rather work with multiple drives than one large drive and several hundred or even several thousand files.

While the primary partition must be located within the first 32 Meg of the hard drive, the extended partition can be located anywhere. FDISK refers to these partitions as "PRI DOS" and "EXT DOS".

Partition sector

The partition sector is the structure that all versions of DOS use to define a hard drive's partitions. When you run the FDISK program for the first time, it creates the partition sector in the hard drive's first sector (cylinder 0, head 0, sector 1).

The BIOS first loads the partition sector, instead of the DOS boot sector, after the system is started or reset. The partition sector is loaded into memory at address 0000:7C00, if there are no diskettes in drive A:.

If the BIOS finds the values 55H, AAH, in the last two bytes of the partition sector's 512 total bytes, it considers the sector to be executable and starts executing the program at the first byte of the sector. Otherwise, the BIOS displays an error message and either starts ROM-BASIC or goes into a continuous loop, depending on the version of BIOS and the manufacturer.

Structure of the partition sector of a hard drive		
Address	Contents	Type
+000h	Partition code	Code
+1BEh	1.Entry in the partition table	16 BYTE
+1CEh	2.Entry in the partition table	16 BYTE
+1DEh	3.Entry in the partition table	16 BYTE
+1EEh	4.Entry in the partition table	16 BYTE
+1FEh	IDcode(AA55h),which identifies the partition sector as such	2 BYTE
Length: 1EH (30) bytes		

This program recognizes and starts the active partition's operating system. To do this, it must load the operating system's boot sector and pass control to the program within that boot sector. Since the later's program code must also be loaded at memory address 0000:7C00, the code from the partition sector is moved to the memory address 0000:0600 first, to make room for the boot sector.

Partition table

The program in the partition sector must be able to find the boot sector for the active partition. For this, it uses the partition table. This table is located at offset 1BEH of the partition sector.

Each entry in the partition table is 16 bytes. The table is located at the end of the partition sector, leaving enough room for 4 entries. So the number of partitions is limited to 4. To accommodate more than four partitions, some hard drive manufacturers use a special configuration program that relocates and enlarges the partition table and adapts the partition sector code to use this relocated table.

Sometimes the partition sector code is changed to allow you to boot any of the installed operating systems on the hard drive. This makes it easy to choose which of the operating systems should run when the computer is first started.

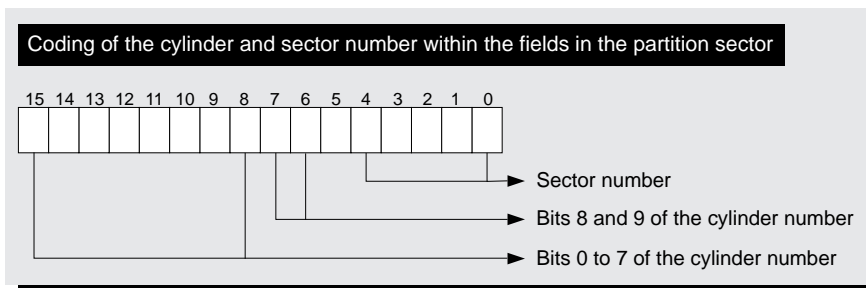
The partition table has the following layout:

Structure of an entry in the partition table		
Address	Contents	Type
+00h	Partition status 00h = inactive 80h = Boot-Partition	1 BYTE
+01h	Read/write head, with which the partition begins	1 BYTE
+02h	Sector and cylinder, with which the partition begins	1 WORD
+04h	Partition type * 00h = Entry not allocated 01h = DOS with 12-Bit-FAT (primary Part.) 02h = XENIX 03h = XENIX 04h = DOS with 16-Bit-FAT (primary Part.) 05h = extended DOS-Partition (DOS 3.3) 06h = DOS-4.0 partition with more than 32 Meg DBh = Concurrent DOS	1 BYTE
+05h	Read/write head, with which the partition ends	1 BYTE
+06h	Sector and cylinder, with which the partition ends	1 WORD
+08h	Removal of first sector of the partition (Boot-sector) of partition sector in sectors	1 DWORD
+0Ch	Number of sectors in this partition	1 DWORD
Length: 10h (16 Bytes)		
* Other codes possible combined with other operating systems or special driver software.		

Starting the boot partition

The first field of each partition table entry indicates whether a partition is active. A value of 00H indicates that partition isn't active; a value 80H indicates that partition is active and should be booted. If the partition sector program detects that more than one partition is active or that none of the partitions are active, it aborts the booting process, displays an error message, and waits in a continuous loop. You can exit this loop only by resetting.

When the partition sector program recognizes the active partition, it uses the next two fields to determine the location of this partition on the hard drive. The sector and cylinder numbers are expressed exactly as BIOS interrupt 13H (Diskette/Hard drive), including bits 6 and 7 of the sector number, which represent bits 8 and 9 of the cylinder number. At this point, BIOS interrupt 13H and its functions are the only way to access the hard drive. The DOS functions aren't available because DOS hasn't been booted yet.



Other information in the partition table

The partition table also contains additional information. For example, each entry has a field that describes the operating system for that partition. The above figure shows the types that are supported.

In addition to the partition's starting sector, another field contains the partition's ending sector, expressed as cylinder, head, and sector numbers.

There are two additional fields for each table entry. The first is the total number of sectors within the partition. The last is the distance of a partition's boot sector from the partition sector, counted in sectors.

Remember the first partition on a drive usually begins in the first sector of track 0, head 1. In other words, almost an entire track is "wasted" because the partition sector occupies only one sector on track 0, head 0.

Structure of the extended partition under DOS

DOS 3.3 allows you to define one primary and one extended partition on a hard drive. FDISK builds the partition sector and partition table to identify the partitions, but doesn't write the program code into the partition sector.

The partition table has two entries. The first entry is for the first logical device of the extended partition and a partition type (value 1 or 4 that indicates a DOS partition with 12-bit or 16-bit FAT). The second entry is for the next logical device within the extended partition, if one exists.

To support other logical devices, this structure is repeated for each additional device. This results in a chained list that continues until the "partition type" field, in the second table entry of the partition table of the partition sector of an extended partition, contains a zero value.

Sample programs to examine the partition structure

You can use the FIXPARTP.PAS, FIXPARTC.C and FIXPARTB.BAS programs to examine the partition structures of a hard drive. FIXPARTP.PAS is written in Pascal and FIXPARTC.C in C. The programs display the contents of the partition sectors and the extended partitions (if there are any) from the hard drive. By default, they access the first hard drive, number 0, but you can also specify a different number (1, 2, 3, etc.) when running the programs to examine a different hard drive.

You'll find the following program(s) on the companion CD-ROM



FIXPARTP.PAS (Pascal listing)
FIXPARTC.C (C listing)
FIXPARTB.BAS (BASIC listing)

15

Accessing And Programming The AT Realtime Clock

DOS uses the date and the time to timestamp files and to provide programs with the current date and time. Many BIOS functions are available to pass this information. Almost all ATs and higher end machines include battery operated realtime clocks. These clocks continue to run when the PC is switched off.

Reading The Date And Time From BIOS

Use BIOS interrupt 1AH to address the various ROM-BIOS time functions. Although the PC and the PC/XT had only two of these functions, the AT and higher end models have eight functions that control the realtime clock (RTC). The AT realtime clock keeps time differently than the older PCs and PC/XTs, which used a software interrupt to perform this task. Before we discuss the BIOS time and date functions, let's look at the earlier method of time control in PC clocks.

Time measurement using timer interrupt 08H

The PC's timer chip (an Intel 8254 or a compatible chip) receives 1,193,180 signals per second from the heart of the system, which is an oscillating quartz crystal. After 65,536 of these signals, or about 18.2 times (18.20648193) a second, the chip calls interrupt 08H. The interrupt controller passes the call to the CPU. Interrupt 08H is called separately from the CPU frequency and takes its frequency from the vibrating quartz crystal. After 18.2 occurrences, BIOS sends interrupt 08H to the ROM-BIOS. This increments a time counter. The current time can easily be calculated in seconds by reading this counter and dividing its value by 18.2. This value can then be converted into hours, minutes, and seconds.

Interrupt 08H also switches off the disk drive motors after a specific period of inactivity. This is done in conjunction with the BIOS interrupt 13H, through which a disk drive can be accessed. Since starting a disk drive motor is time-consuming, the motor remains on after receiving a BIOS disk function so it won't have to be restarted for the next disk operation.

After the interrupt interface has completed its tasks, it calls interrupt 1CH. Usually this interrupt carries an IRET instruction, which immediately returns program execution to the time interrupt interface. However, a program may define its own interface to use the cyclical time signal. This is useful for permanently displaying the current time on the screen, which is demonstrated by a TSR program in Chapter 34.

Timer interrupt 08H is the only way to measure time in PCs and PC/XTs. The AT and higher end computers contain realtime clocks with battery backup, which continue measure the time without a timer interrupt. The AT ROM-BIOS still manages the timer interrupt, however, to maintain downward software compatibility to the PC and PC/XT systems.

Function 00H:	Get clock
---------------	-----------

Function number 00H obtains the current clock time. You can call this function by passing the number 00H to the AH register. The function loads the time into the CX and DX registers. These two registers combine to form a 32-bit counter value (CX contains the most significant 16 bits, while DX contains the least significant 16 bits). The BIOS timer increments this value by 1 each time interrupt 8H is called (18.2 times per second). The total value is the result of multiplying the contents of CX register by 65,536 and adding the contents of the DX register. Dividing this value by 18.2 returns the number of seconds elapsed, which can then be converted into minutes and hours.

The AT interprets time differently than the PC and XT. The PC/XT BIOS sets this counter to 0 during the system booting process. The value returned is the time passed since the computer was switched on (not the actual time). To obtain the time, the current time must be converted to the value corresponding to the counter, then passed to the BIOS (more on this later). The AT doesn't require this time value conversion because BIOS reads the actual time from the realtime clock during the

system boot. It converts this time into a suitable timer value and then saves it. Reading the counter with the help of function 0 on the AT provides the current time. Besides this counter, a value in the AL register indicates whether 24 hours have passed since the last reading. If the AL register contains a value other than 0, 24 hours have passed. This value doesn't indicate how many 24-hour periods have elapsed since the last reading.

If the conversion of time values into clock time is too complicated, function 2CH of DOS interrupt 21H can be used. This function simply reads and converts the current time using function 0 of interrupt 1AH. (See Chapter 22 for more information about function 2CH of DOS interrupt 21H.)

Function 01H:	Set clock
---------------	-----------

Function number 01H sets the current clock time. Call this function by loading the number 1 into the AH register, the most significant 16 bits of the counter into the CX register, and the least significant 16 bits into the DX register. These two registers combine to form a 32-bit time value. If the conversion of the current time into a timer value is too complicated, function 2DH of DOS interrupt 21H can be used instead (see Chapter 24 for more information about function 2DH of DOS interrupt 21H.)

Functions for accessing the realtime clock

The following six functions are available only on the AT and higher end models. Although realtime clocks are also available for the PC and the PC/XT, they usually aren't supported by the computer's ROM-BIOS. So, unless the manufacturer also supplies a TSR program that implements the corresponding AT BIOS functions, calling these functions won't return any results on PCs and PC/XTs. When developing a program that accesses AT clock functions, test the model identification byte in F000:FFFE to ensure that the system is an AT or higher end machine (see Chapter 3 for more information about the model identification byte).

All six functions use BCD format for time and date indications. In this format, two characters are coded per byte; the higher number is coded in the higher nibble and the lower number in the lower nibble. All six functions use the carry flag following a return from the function call. If the carry flag is set, this indicates that the realtime clock is malfunctioning (e.g., from a dead battery). The called function couldn't be executed properly.

Function 02H:	Get current time
---------------	------------------

Function 02H reads the realtime clock time. You can call the function by loading the function number (2) into the AH register. The current time is returned with the hour in the CH register, minutes in the CL register, and the seconds in the DH register.

Function 03H:	Set current time
---------------	------------------

Function 03H sets the time on the realtime clock. You can call this function by loading the function number (3) into the AH register, the hour into the CH register, minutes into the CL register, and seconds into the DH register. The DL register indicates whether the "daylight savings time" option should be used. A 1 in the DL register selects daylight savings time, while 0 maintains standard time.

Functions 04H and 05H read and set the date stored in the realtime clock. Both functions use the century, year, month, and day as arguments. The day of the week (also administered by the realtime clock) doesn't apply to these functions. If you want to read the day of the week, you must directly access the realtime clock.

Function 04H:	Get current date
---------------	------------------

Function 04H gets the current date from the realtime clock. You can call this function by loading the function number (4) into the AH register. The CH register contains the first two numbers of the year (the century). The CL register contains the last two numbers of the year (e.g., 88). The month is returned in the DH register and the day of the month in the DL register.

Function 05H:	Set current date
---------------	------------------

Function 05H sets the current date in the realtime clock. You can call this function by loading the function number (5) into the AH register, either 19 decimal or 20 decimal into the CH register, the last two numbers of the year into the CL register (e.g., 89 decimal), the month into the DH register, and the day of the month into the DL register.

Function 06H:	Set alarm time
---------------	----------------

Function 06H allows the user to set an alarm. Since only the hour, minute, and second can be indicated, the alarm time applies only to the current day. When the clock reaches the alarm time, the realtime clock calls a BIOS routine that in turn calls interrupt 4AH. A user routine can be installed under this interrupt to simulate the sound of an alarm clock. (You can program the routine to make other sounds.)

During the system initialization interrupt 4AH moves to a routine that contains only the IRET assembly language instruction. The IRET instruction forces the CPU to terminate the interrupt so that arriving at alarm time doesn't result in any action visible to the user. You can call this function by loading the function number (6) into the AH register, the alarm hour into the CH register, the alarm minute into the CL register, and the alarm second into the DH register.

Function 07H:	Reset alarm time
---------------	------------------

Only one alarm time can be set. If this function is called while another alarm time is set or hasn't been reached yet, the carry flag is set after the function call. A new alarm time doesn't replace the old alarm time; the old time must be deleted first. You can call this function by loading the function number (7) into the AH register; no other parameters are required. This call clears the last alarm time so that a new alarm time can be programmed.

To call this function, place its function number in the AH register; no further arguments are required. This function call will then delete the current alarm time so a new alarm time may be set, if desired.

Reading And Setting The Realtime Clock

The AT and higher end models each have a battery operated realtime clock on the main circuit board. This clock is part of the Motorola MC-146818 processor, which contains 64 bytes of battery backup RAM. This RAM accepts clock data and system configuration data. It can be accessed through port addresses 70H to 7FH. However, the user needs only ports 70H and 71H.

Realtime clock registers

As the following table shows, the clock has sixteen important memory registers:

Realtime clock registers			
Register	Meaning	Register	Meaning
00H	Current second	08H	Month
01H	Alarm second	09H	Year
02H	Current minute	0AH	Clock status register A
03H	Alarm minute	0BH	Clock status register B
04H	Current hour	0CH	Clock status register C
05H	Alarm hour	0DH	Clock status register D
06H	Day of the week	32H	Century (19 or 20)
07H	Number of day		

Each time field (second, minute, hour) has a similar alarm field. With these alarm fields, a programmer can set the clock to trigger an interrupt at a particular time of the current day (more on this later).

Weekday

The day of the week provides the number of the current weekday. The value 1 represents Sunday, the value 2 represents Monday, 3 for Tuesday, etc.

Year

The year is counted relative to the century (the system assumes 1900). The value 87 in this field represents the year 1987. The four status registers enable the user to program the clock.

Accessing the individual registers of the realtime clock

Since the registers are a part of the 64-byte RAM, you can access them like any other memory location. First load the number of the memory location to be accessed into the AL register. Then pass this value to port 70H using the OUT instruction. Since direct value output to a port is impossible, register AX is used for the output. The chip recognizes that an access to one of its memory locations occurred. Then either an OUT instruction writes to port 71H or an IN instruction reads the memory contents from port 71H. The following instructions read or write a memory location in the realtime clock:

READ:

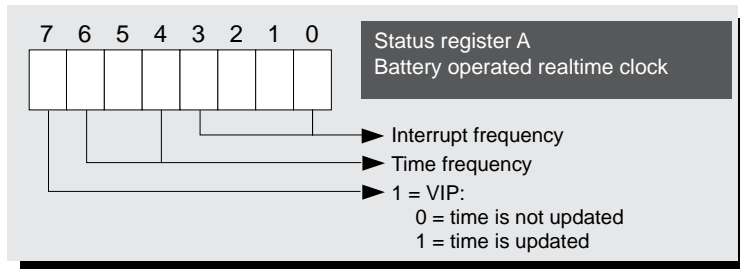
```
mov    al,Memory_location
out    70h, al
in     al,71h
```

WRITE:

```
mov    al,Memory_location
out    70h,al
mov    al,New_contents
out    71h,al
```

Status register A

The four status registers are of particular interest to us because these registers are used for programming the clock. The following is a description of status register A:



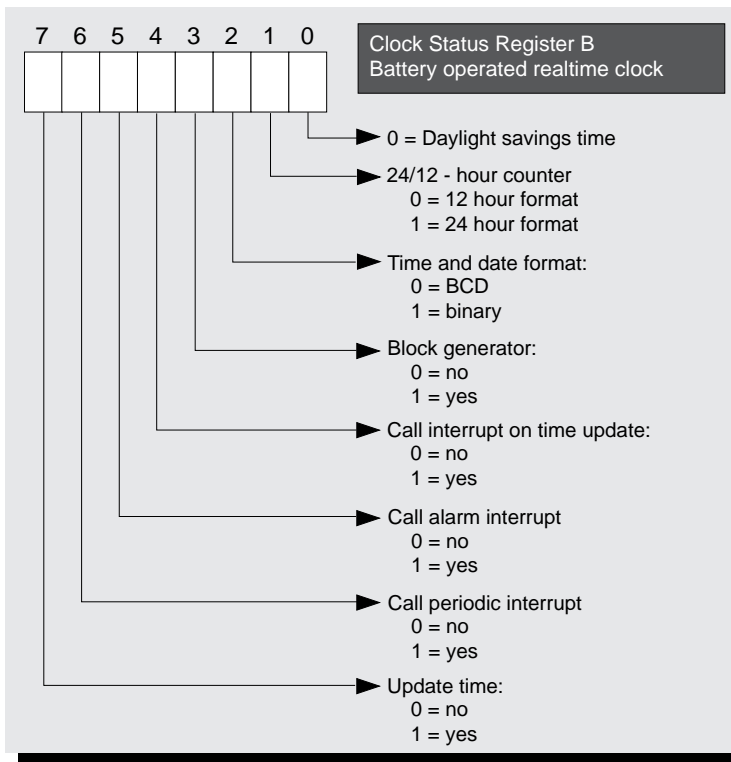
The ROM-BIOS set the two lower fields of these registers during the system boot. The interrupt frequency field has a default value of 0110b. This value results in an interrupt frequency of 1024 interrupts per second (i.e., an interrupt every 976,562 microseconds).

The contents of the time frequency field is 010b. This field triggers a time frequency of 32,768 KHz.

Bit 7 of the status register is used with these two fields. This bit indicates whether a second has just elapsed and increments the time fields (seconds, minutes, hours). If a second hasn't elapsed, this bit contains a 1. With this bit, you can read the individual time fields only when the time isn't being updated. Otherwise, a minute could pass and the second counter reset to 0 before the minute counter could be incremented. This could cause a time jump from 13:59:59 to 13:59:00, then the correct display of 14:00:01 one second later.

Status register B

Several clock parameters can be programmed through status register B:



Some clock settings can be programmed through status register B. Bit 0 of status register B controls daylight savings time status. When this bit is set to 1, it indicates that daylight savings time is in effect. A value of 0 (the default value for this bit) indicates that standard time is in effect. Bit 1 determines whether the clock should operate in 12-hour or 24-hour mode. In 12-hour mode, the clock switches to 1 o'clock after every 12 hours (midnight and noon). The 24-hour mode switches to 1 o'clock after 24 hours. This mode is active when you boot the system.

Bit 2 defines the format in which the time and date fields are stored. If this bit contains a 1, the various dates are stored in binary notation. The year (19)87 is coded as 01010111b in BCD format, which is switched on by the value 0 in bit 2. Two numbers are stored in every byte. The higher half is stored in the most significant four bits and the lower half is stored in the least significant four bits. Usually this bit contains a 0 and the numbers are stored in BCD format.

Bit 4 determines whether an interrupt should be called after the time (and date) update. This bit must contain a 1 if an interrupt should be called. The system suppresses this interrupt by setting this bit to 0 during the booting process. Bit 5 can trigger an alarm. The clock reads the alarm time from locations 1, 3, and 5 (seconds, minutes, and hours) of clock RAM. When the alarm time is reached, an interrupt executes when bit 5 is set to 1. The system suppresses this interrupt when it sets bit 5 to 0 during the booting process.

Bit 6 controls periodic interrupt calls when it is set to 1. The frequency of the interrupt calls depends on the interrupt frequency coded into bits 0-3 of status register A. Since the default value on bootup is a frequency of 1,024 KHz, the interrupt triggers every 967,562 microseconds. Since bit 6 is set to 0 at the system start, an application program must set it to 1 before periodic interrupt calls can execute.

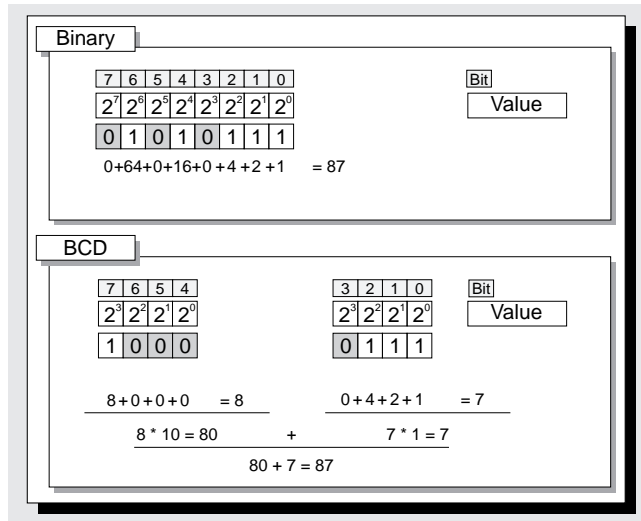
Bit 7 controls the periodic updating of the time and date, once every second. This bit is set to 0 when you boot the

NOTE

BIOS assumes BCD representation when performing the date function with interrupt 1AH. Programs that call these functions and obtain the information in binary format instead of the expected BCD may crash. The same applies to the 12-hour/24-hour time measurement, although a change to the 12-hour cycle wouldn't result in as serious consequences as the change in the date.

system so that the time constantly increments. Before entering a new date and time in the various memory locations, this bit should be set to 1 first to prevent the clock from changing the time immediately. Once you've entered all the necessary data, this bit can be reset and the time can continue updating.

*The number 87
in binary and
BCD format*

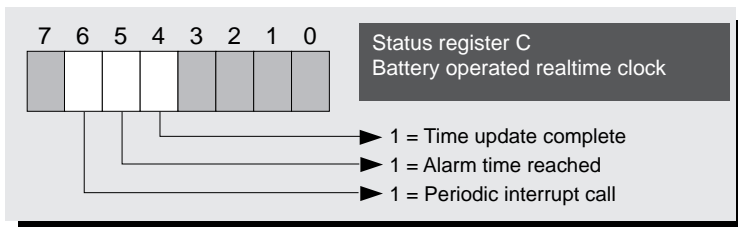


RTC interrupt 70H

While discussing the bits in status register B, we've repeatedly referred to interrupt 70H. The realtime clock calls this interrupt under certain conditions. Even though there are several reasons for the clock to call an interrupt (alarm time, periodic interrupts, etc.), interrupt 70H is consistently called. This interrupt contains a BIOS routine that controls the two time functions in interrupt 15H, among other things.

Status register C

The routine uses status register C of the clock to determine the reason for the call. Only bits 4, 5, and 6 of this register are important to us at the moment. These bits correspond to the bits in status register B. For example, when you trigger the alarm interrupt (which can only occur if bit 5 in status register B was set), then bit 5 in status register C is also set to indicate that the alarm time has been reached.



The first task of the routine that intercepts interrupt 70H is to read status register C. This routine then determines the reason for the interrupt call and reacts accordingly.

Status register D

Bit 7 is the only important bit in status register D. This bit indicates the status of the battery that maintains the data storage even when the PC's power supply is switched off. If this bit has the value 0, you should replace the battery because the present battery is dead or almost dead.

Some configuration data follows status register D.

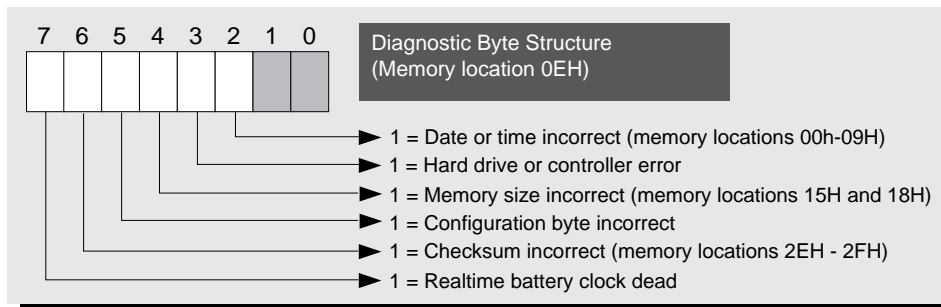
Configuration Data And Battery Operated RAM

Besides the date and time information, the 64 battery backed memory registers also contain configuration data. Of the memory locations of various BIOS manufacturers, only those that are designated as unreserved contain the same information. Since all the other locations are used at the discretion of BIOS and hardware designers, these locations shouldn't be overwritten by a program.

Battery backed RAM registers			
Addr.	Content	Addr.	Content
0EH	Diagnostic byte (see below)	18H	High byte in K of an expansion board's main memory size
0FH	Status at system power-down	19H	Reserved
10H	Write to disk (see below)	2DH	Reserved
11H	Hard drive 1 type	2EH	Checksum high byte (memory locations 10H - 2DH)
12H	Hard drive 2 type	2FH	Checksum low byte (memory locations 10H - 2DH)
13H	Reserved	30H	Low byte in K of expansion memory size
14H	Configuration byte (see below)	31H	High byte in K of expansion memory size
15H	Low byte in K of hard drive main memory size	32H	First two century digits in BCD notation
16H	High byte in K of motherboard main memory size	33H-3FH	Reserved
17H	Low byte in K of an expansion board's main memory size		

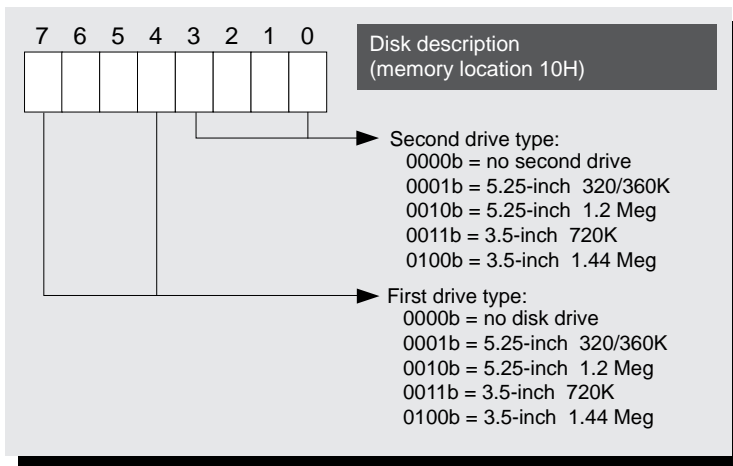
Diagnostic byte (0EH)

The diagnostic byte documents various errors that may occur during the Power-On Self Test (POST).



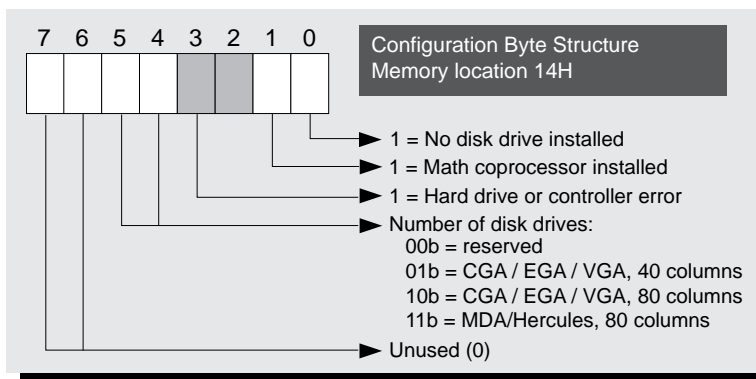
Disk description (10H)

Memory address 10H of the battery backed RAM contains information identifying the first and the second disk drive formats (5.25-inch or 3.5-inch) and their capacities.



Configuration byte (14H)

Memory address 14H of the battery backed RAM contains configuration data that specifies the number of disk drives, the video mode at system startup, and the availability of a math coprocessor.



Sample programs to access the realtime clock

The RTCB.BAS, RTCP.PAS and RTCC.C programs on the companion CD-ROM show how you can access the realtime clock from BASIC, Pascal, or C. Three routines perform most of the functions. The first routine reads a value from one of the clock's memory locations. The second routine places a value there. The third routine checks whether the clock is operating in binary mode or BCD mode, then reads a memory location in the clock, converting the contents of this location from BCD into binary if necessary. This routine is important for accessing all memory locations, containing information on date and time, that could be coded in BCD or in binary format.

The main program checks the battery on the clock. If there's power in the battery, the program calls two routines that, among other things, read the contents of the memory locations for the current date and current time from the clock. This data appears on the screen.

The main program doesn't access the routine for a description of memory locations. However, you can easily convert the program so the routine for the description of memory locations writes to the clock instead of reading date and time. This is only a suggestion; feel free to experiment.

You'll find the following program(s) on the companion CD-ROM



RTCB.BAS (BASIC listing)
 RTCP.PAS (Pascal listing)
 RTCC.C (C listing)

16

System Configuration And Processor Types

Knowing the configuration of a computer is often important. A program must frequently know the number of serial ports, the size of the RAM memory or whether a math coprocessor is present. In this chapter we'll show you how to determine various configuration data with the help of two ROM-BIOS interrupts. You'll also learn how to determine the processor type and coprocessor type of a PC.

Determining The Configuration Using The BIOS

The ROM-BIOS has two interrupts for reading configuration information:

- Interrupt 11H
- Interrupt 12H

Each interrupt contains one function. With these interrupts you can obtain information about the hardware environment of a PC and determine the size of the RAM memory.

Unfortunately, even after using these interrupts, many of your questions will still be unanswered, as the following two sections will demonstrate.

Reading the hardware environment

Call BIOS interrupt 11H to obtain information about the hardware environment of a computer, the number of serial and parallel ports, disk drives and the DMA controller.

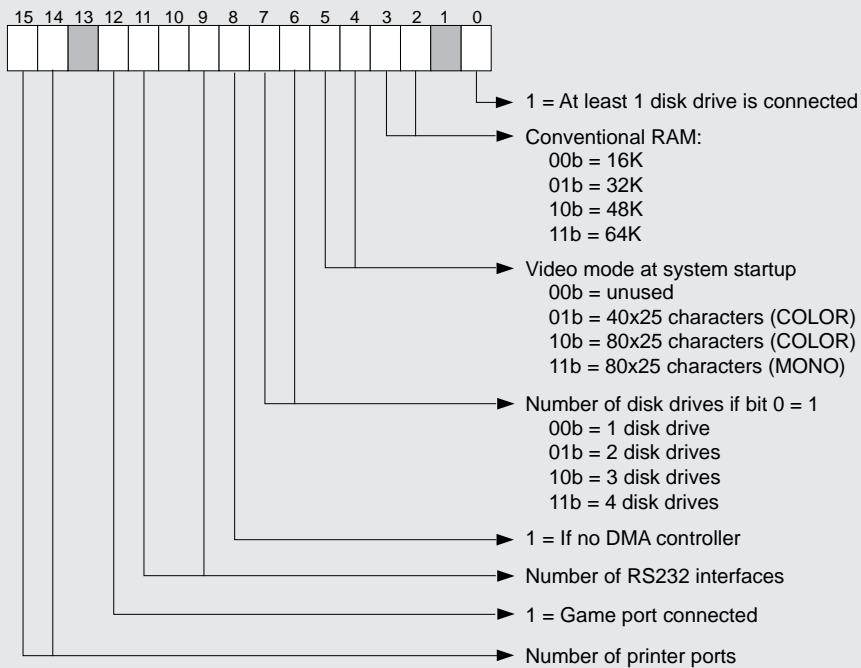
Unlike many other BIOS interrupts, the contents of the processor register are unimportant at the time this interrupt is called because a function number or any other argument isn't necessary. This interrupt consists of only a single function, which returns the desired information about the configuration in the AX register.

The illustration at the top of the following page shows the contents of the AX register after the function call represent a bit field. These fields provide information about various components of the hardware. However, this information is limited. Since this interrupt was introduced with the first PC and its original BIOS, it reflects the modest hardware environment that is typical of the early PCs.

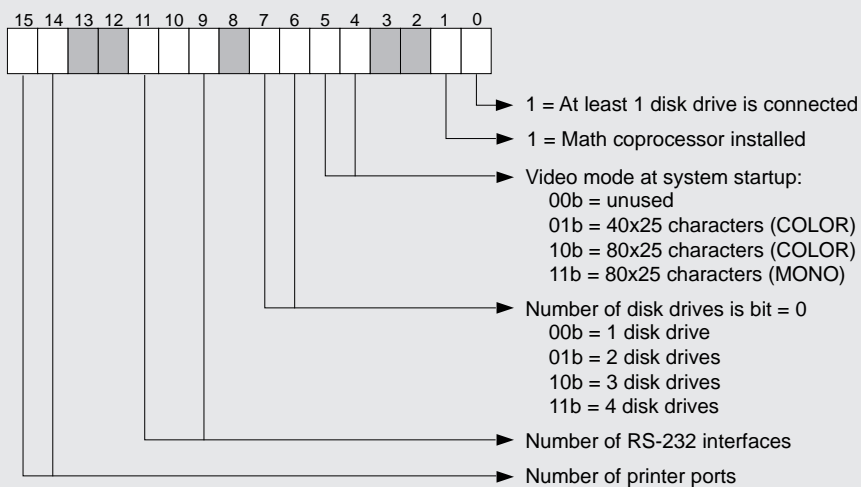
Although the bit assignment above applies to both PCs and XT's, it differs from the structure of the configuration word returned by the AT and its descendants. When the AT was introduced, this interrupt was revised to use the advancements in hardware technology and the improved PC equipment.

However, this interrupt doesn't provide important information, such as the processor type, the kind of keyboard, and the presence of a mouse or EMS memory. So, if you want to determine whether a mouse is present or what kind of video card is installed, refer to the appropriate chapters in this book.

BIOS configuration information (PCs and XTs)



BIOS configuration information (ATs)



Don't use this function to prompt for the current video mode because it only specifies the video mode that was switched on when the system was running. You should use function 0FH of interrupt 10H, which returns the number of the current video mode. We'll talk about a sample program which demonstrates how to use this interrupt later in this chapter.

Determining RAM configuration using the BIOS

Interrupt 11H, which hasn't changed since 1981, provides only the size of the RAM memory on the motherboard. However, calling interrupt 12H provides the size of the entire RAM memory. When the system boots up, the size of the RAM on the motherboard and any existing memory expansion boards are added up and saved. On PCs and XTs, this information is taken from the settings of the single DIP switches on each memory board, while on ATs the information is read from one of the 64 memory locations of the battery operated clock. However, this method only determines the size of the RAM memory below the 1 megabyte boundary. This is sufficient for PCs and XTs because the address space of their processors is limited to 1 megabyte. So, they cannot have any additional RAM memory.

ATs and their descendants, however, don't fit into this category; they have processors capable of managing up to 16 megabytes of memory. On these computers you can install additional memory, which interrupt 12H cannot detect. So you should use function 88H of BIOS interrupt 15H instead. This function reads the size of the memory above the 1 megabyte boundary. Like interrupt 11H, interrupt 12H doesn't expect any arguments in the various processor registers when you call it. It returns the memory size in the AX register as a factor of 1K (1024 bytes instead of 1000 bytes).

Sample programs

Three programs in BASIC, Pascal and C on the companion CD-ROM should demonstrate how to use the interrupts we just discussed. The three programs have the same structure and output the configuration data that can be determined by using BIOS interrupts 11H and 12H.

These programs first read the model identification byte in memory location F000:FFFE to determine whether the computer is a PC, XT, or AT. This information forms the foundation for the remaining procedures of the program. Then interrupt 12H reads the size of the RAM memory on the motherboard and outputs this information. On ATs, the size of the RAM memory beyond the 1 megabyte boundary is determined with the help of function 88H of BIOS interrupt 15H.

After this information is output, the program calls interrupt 11H, which determines the configuration of the computer. The program's final task involves filtering the information from the various bit fields. To keep the program as short as possible, we only used bits that are identical in both the configuration word of the PC/XT and the AT. For example, on ATs, the information about whether the computer has a mathematical coprocessor isn't used. This information isn't always reliable and the programs from the next section are much better at determining it.

Determining Processor And Coprocessor

Many programs are available which provide information about the configuration or layout of a PC. Besides the size of the RAM memory, the installed version of DOS, etc., you'll also learn which processor the computer uses and which coprocessor assists this processor.

This information can be very important, for example, when developing programs in high level languages because code generation can be adapted to a particular processor. Both the Microsoft C compilers and their Borland counterparts allow special code generation for the 8088, the 80286, or the 80386. This enables you to fully use the processor's capabilities and the command set. Generally this noticeably increases the performance of programs that work with large data sets.

One way you can use this capability is by compiling each of the processor types separately. Also, you can develop a program, which is used as a loader for the actual program, that prompts for the processor type after starting. Then it executes the particular program (of the three programs) that was compiled for the relevant processor. You can also do this with the numerical coprocessor.

However, now we must decide how to determine the installed processor. Unlike other configuration data, this cannot be determined by calling a BIOS or DOS function. Also, since there isn't a corresponding machine language command, which causes the

You'll find the following program(s) on the companion CD-ROM



CONFIGB.BAS (BASIC listing)
CONFIGP.PAS (Pascal listing)
CONFIGC.C (C listing)

processor to reveal its identity, you must use a complicated procedure that, according to statements of the hardware manufacturers, shouldn't even work.

This procedure involves a test that's based on how certain machine language commands are executed. Although the processors from the 8086 to the Pentium are software compatible, small changes were made in the logic of some commands during the development of this processor series. Since these changes are noticeable only in rare situations, a program developed for the 8088 will also run on all other processors of the Intel 80xxx series. So if you intentionally put the processor in this type of situation, you can establish the identity of the processor as a result of its behavior.

Since these differences are only noticeable on the machine language level, you can write such a test program only in machine language or assembly language. At the end of this chapter, this type of test routine is presented in the form of two assembly modules for linking to Pascal and C programs. We'll discuss how they operate and the differences between the individual processors in the following sections.

Determining the processor type

The test routine for determining the installed processor consists of several tests that can be used to differentiate the individual processor types. A test will run only if the preceding test fails.

Varying layout of the flag register

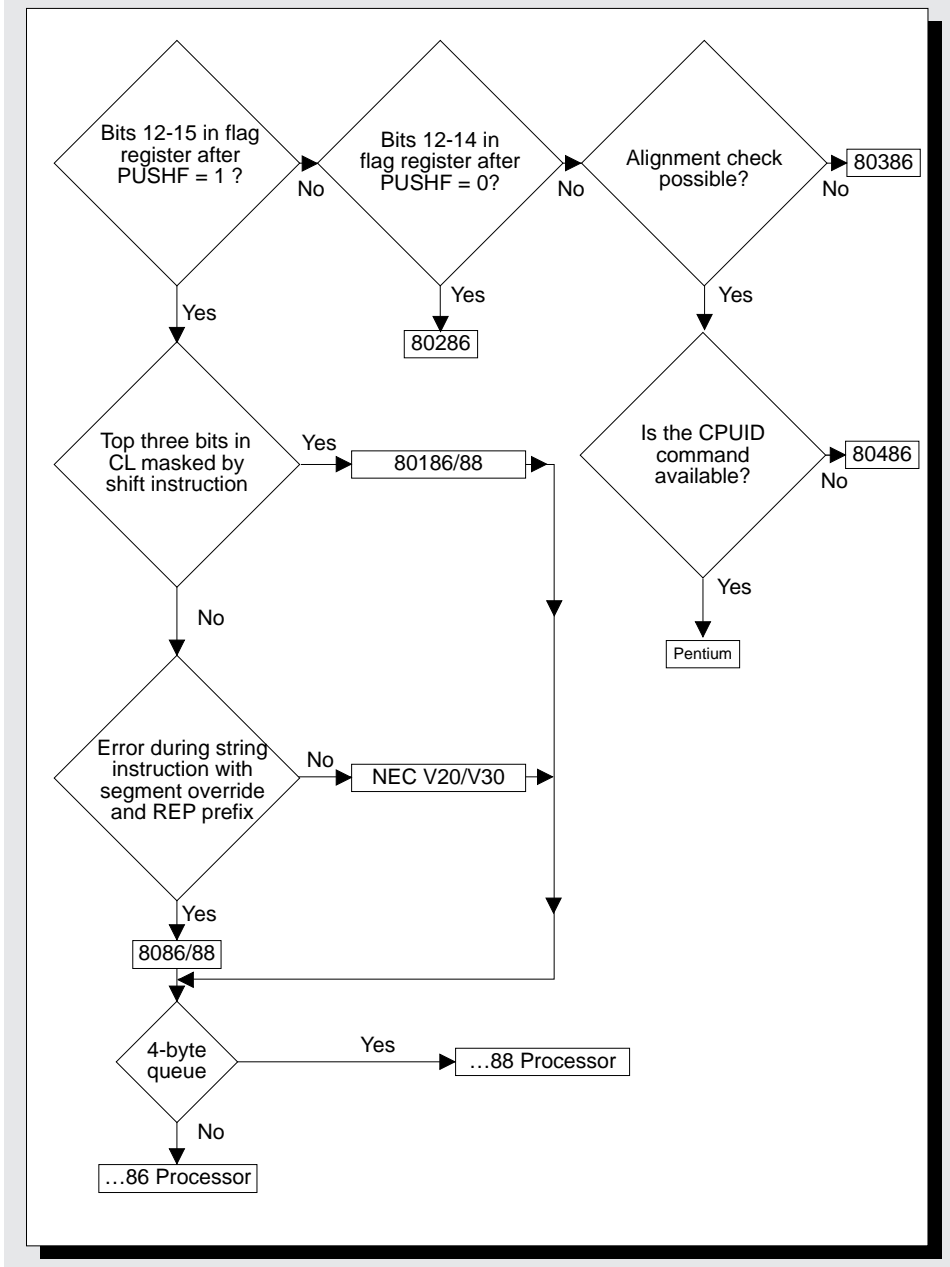
The first test is based on the varying layout of the flag register on the different processors. Although the meaning of bits 0 to 11 are identical for all processors, bits 12 through 15 don't have a meaning until the 80286 (through the introduction of Protected Mode). The commands PUSHF (place contents of flag register on stack) and POPF (retrieve contents of flag register from stack) take this circumstance into account. These commands always set bits 12 to 15 of the flag register to 1 on all Intel processors up to the 80188. Beginning with the 80286, the processors behave differently.

The first test uses this by placing the value 0 on the stack and then using the POPF command to fetch it back to the flag register. Since there isn't a command for reading the contents of bits 12 through 15 in the flag register, the PUSHF command immediately places the flag register on the stack. However, this only happens so the POP AX command in the AX register can send for the flag register. It's easy to test the layout of bits 12 to 15 in the AX register. If all four bits are set, it cannot be a 80286 or any of its descendants, and the routine continues with the next test.

If all four bits aren't set, then the list of possible processors is limited to the 80286 and its descendants. Since the 80286 processes the POPF command differently than the 80386/80486, it's easy to distinguish the 80286 from its descendants. The entire procedure is repeated again, with the value 7000H, instead of 0, being placed on the stack. When the POPF command loads the flag register, it causes bits 12 to 14 to be set to 1.

However, if these bits no longer contain the value 1 when retrieving the flag register from the stack, the processor must be an 80286, which automatically sets these three bits to 0, unlike its successors. If this test indicates the processor is an 80286, the test routine ends.

Determining the processor type



Distinguishing the 80386 and 80486

If the processor isn't an 80286, you must determine whether it's an 80386 or an 80486. Again, you can use the flag register to do this because the 80486 has a new flag in the extended flag register EFlags. This flag, which is called "Alignment Check", is located at bit position 15. When this flag is set, the processor checks all memory accesses with the specified offset address. If the address isn't a multiple of four, the processor releases a special interrupt, called an Exception.

If the Alignment Check is operating, this will be the fate of the subsequent machine language command, for example, because it reads out the contents of memory location 102. Since 102 isn't a multiple of four, it isn't aligned. This also applies to memory location 77, which is addressed in the following command:

```
mov  al,[102]
shl  word ptr [77],3
```

Both of these commands would execute an alignment exception, which causes an exception handler to be executed the same way as an interrupt. The operating system usually intercepts this exception handler to acknowledge the misalignment. Depending on the user's judgment, the program can either abort or continue executing the command.

Access to memory on the 80486 is especially fast when the data to be processed is located at an offset address that's divisible by four. So, if an offset address isn't divisible by four, memory access on the 80486 takes a long time. To prevent this from happening, an operating system can enable the alignment check so it's aware of programs that access data which are misaligned during execution.

The operating system considers the release of the exception as an opportunity to send a warning message to the developer. In this way, the alignment check can help optimize programs for the 80486.

The results of the alignment check make it easy to distinguish between the 80386 and the 80486. Since the 80386 isn't familiar with the Alignment flag, it doesn't change its contents. So, you should first read and save the previous contents of the EFlags to reset the alignment bit. Then read the EFlags register again to check whether the contents of this flag have actually changed. If the contents have changed, you know that you're working with an 80486.

However, remember to set the Alignment flag back to its original value when you're finished. Since the EFlags register can be loaded and manipulated only from the stack, there are some other dangers. The following machine language commands demonstrate this process, which loads the EFlags register into the EAX register. This first command "pushes" the contents onto the stack so they can be "popped" into the AX register.

```
pushfd
pop  eax
```

Although it isn't evident by looking at the two commands, memory is accessed twice because this is where the stack is located. If the alignment bit is already set when this command is executed, an exception is triggered if the stack pointer SP doesn't refer to an address that's divisible by four. However, this can easily be avoided by placing the following machine language command before the previous commands:

```
and  esp,0FFFCh
```

This command simply sets the two bottom bits in the ESP register to 0, rounding off the stack pointer to the next address divisible by four. This type of processor test is usually executed within a subroutine. So, you should record the current contents of the stack pointer first so you can restore them later. Otherwise, at the end of the procedure, the stack pointer won't point to the return address of the caller, which was placed there when the subroutine was called. As a result, the program crashes.

Reading the SX versions

Although this test differentiates between 80386s and 80486s, it doesn't enable you to distinguish between the DX and SX versions of these processors. Also, no software programs distinguish between the 80386SX and the 80386DX.

However, it's possible to distinguish between the 80486SX and the 80486DX, even if only indirectly. The SX version of this processor doesn't have the mathematical coprocessor 80387, which is integrated in the DX version. Once you determine that you have an 80486 processor, you can easily determine whether the corresponding floating point commands are available. To do this, use a coprocessor test, which allows you to indicate the installed version of the processor.

However, just because floating point commands are supported doesn't mean that an 80486DX is present. It could also be an 80486SX that has a mathematical coprocessor. You can assume that you have an 80486SX only when you encounter an 80486 that doesn't support numerical operations of the floating point processor.

Differentiating between the 80486 and the Pentium

Regardless of whether you are dealing with an SX or a DX, after ruling out the possibility of the machine being an 80386, you cannot jump to the conclusion that you have an 80486. The misalignment flag is also supported by successors of the 486. This includes the Pentium. So, the final test must serve to differentiate between the 80486 and the Pentium.

You need to use a flag from EFLAGS register again. This time we'll use bit 21. Since it was introduced with the Pentium, it's not available for the 80486. The Pentium and its successors use this bit to indicate their support for a new machine language command for CPU identification: The CPUID command.

To check if this flag exists, first call the EFLAGS register using the stack to the EAX register and note it in the ECX register. Then rotate the contents of bit 21 and write the new value back to the EFLAGS register using the stack. Now you can determine if you're using an 80486. All you have to do is use the stack to load back the EFLAGS register in the EAX register. Next, compare the new contents with the old contents stored in ECX. If the contents match, it means bit 21 could not be changed. This proves a 486 exists because it is possible to set this bit with the Pentium and its successors.

Distinguishing between 8086/8088 and 80186/80188

If the processor failed the first test, then the processor must be one of the four ancestors of the 80286. The following test determines whether the processor is an 80188 or 80186.

When these processors were introduced, the operation of the various shift commands (e.g., SHL and SHR) were changed to work in connection with the CL register as a shift counter. In the previous processors, the contents of the CL register specified the number of shifters in a range between 0 and 255. However, in the new processors the top three bits of the CL register are deleted before the shift, which limits the maximum number of shift operations. This makes sense because after 16 shifts of a word (17 if shifting by the carry flag), all bit positions contain the value 0. Additional shifting wastes valuable processor time, but no longer changes the value of the word.

The test considers this behavior by using the SHR command to shift the 0FFH value in the AL register by 21H positions to the right. If the processor is an 80188 or newer, it will first mask the upper 3 bits in the shift counter and then leave only a single shift of the 21H shifts.

Number of shifts	021H (00100001b)
Mask the upper 3 bits	& 01FH (00011111b)
Actual number of shifts	001H (00000001b)

Unlike its predecessors, which actually shift the value FFH 21H times to the right, returning the value 0, the 80188 and 80186 return the value 7FH as a result. By examining the AL register after the shift, it's easy to determine whether the processor is an 80186 or 80188 (AL doesn't equal 0) or neither (AL equals 0).

Differentiating between 8086/8088 and NEC V20/NEC V30

If the processor didn't pass this test either, it must be an 8088, an 8086, or a clone from NEC. These clones, which are labeled V20 or V30, are the most common clones of the two Intel processors. They have the same command set as the Intel processors they're modeled after, but have a higher processing speed due to optimizations in the inner logic.

In addition to these improvements, the small error which occurred in some of the 8088 and 8086 processors, was eliminated in these processors. This error occurs if a hardware interrupt is triggered during the execution of a string command (e.g., LODS) in connection with the REP(eat) prefix and a segment override. After the interrupt ends, the command doesn't resume execution on some 8088 and 8086 processors. This is indicated by the fact the CX register, which functions as a cycle counter for this command, doesn't contain the value 0, as expected, after execution of the command.

The test program first loads the value 0FFFFH into the CX register and then executes the subsequent string command, with the REP prefix and segment override, 65,535 times. This process is time-consuming even for a fast processor. So, during one

of the 65,535 executions of this command, a hardware interrupt will be triggered. In the case of the 8088 and 8086, the command isn't resumed and the remaining "loop runs" aren't executed. The test program uses the CX register to check this after the command is executed.

Reading the width of the data bus

The last test is for processors that are predecessors to the 80386. In this test, you determine whether the processor has an 8 bit or a 16 bit data bus. In other words, you're finally going to discover whether the identified processor is an 8088 or 8086, a V20 or V30, or an 80186 or 80188. Although you cannot use machine language commands to determine the width of the data bus, the length of the prefetch queue within the processor is related to this size.

The prefetch queue, which is located inside the processor, is responsible for loading the machine code of the subsequent commands into the processor while the current command is being executed. This can significantly increase the execution speed of the processor because the mechanism for filling the prefetch queue works parallel to the actual execution of the command. This allows the processor to access the next command immediately after one command ends. The processor doesn't have to worry about the address bus and data bus for loading the command.

As long as it doesn't use self-modified code, a program usually doesn't see the prefetch queue being filled. This refers to changes to the program code that were made by the program (e.g., by overwriting one machine language command in the code segment with another one from the program code). However, this isn't an unusual technique.

Problems occur with this method when commands, whose machine codes have already been loaded into the prefetch queue, are changed. This occurs when the command to be modified is very close to the command that's modifying it. For example, in the following code excerpt, an INC-CX command is replaced by an NOP command. The INC command is still executed because the two commands are so close to each other the modified INC command was already in the prefetch queue at the time of its execution.

```
        mov byte ptr cs:queue,Code NOP
queue:  inc dx
```

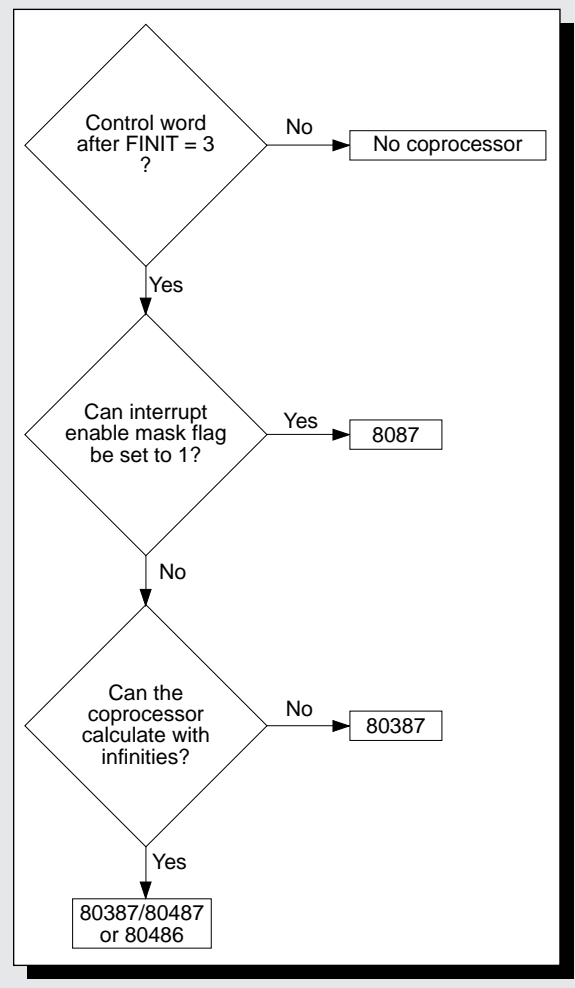
This procedure simplifies testing the length of the prefetch queue. With this information, you can also determine the width of the data bus. This is possible because processors with a 16 bit data bus have a 6 byte prefetch queue, while processors with an 8 bit data bus have 4 byte prefetch queues.

Test the length of the prefetch queue by using the STOSB (store string byte) string command to modify 3 bytes in the code segment. These three bytes appear immediately after the STOSB command. In this test, the bytes must be placed so they are already located within the queue on a processor with a 6 byte queue. This must be done so the processor doesn't notice this change to the program code. However, on a processor with a 4 byte queue, these commands are still outside the queue. So the changed commands are loaded and executed in their modified form the next time the queue is "filled up".

The program takes advantage of this because the INC DX command is among the modified commands. This command increments the contents of the DX register. Since this register receives the code of the determined processor within the test routine, it causes a switch to the next processor type. However, this happens only if this command is actually executed and if it's a processor with a 6 byte queue.

Since this always increases the processor code by 1 on xx86 processors, the processor codes within the test routine are chosen so the code for the corresponding xx88 processor is always followed by the code for the matching xx86 processor. For example, if processor code 4, for the 80188, is loaded into the DX register before the beginning of this test, then the processor code within this test is incremented by 1. The result is processor code 5, in the DX register, which is interpreted as a code for the 80186 by the test program.

Determining the presence of a coprocessor



Coprocessor test

The installed coprocessor is also important although less than three percent of all computers have them. However, often just knowing that a mathematical coprocessor isn't installed is valuable information.

You can easily check whether a mathematical coprocessor is installed by using the machine language command FSTCW command. Before using this command, reset the coprocessor with the FINIT machine language command. FSTCW deposits the control word of the coprocessor into a specified variable and the hi byte of this register will always contain the value 3 after you reset the coprocessor. You'll immediately know whether there is a coprocessor because you won't be able to execute the FSTCW command and the value 3 won't be in the specified memory location unless you saved this value earlier.

Tests for determining the coprocessor

After discovering that your computer has a coprocessor, you must determine the type of coprocessor. Since the integrated coprocessor in the 80486 is identical to the 80387, you must simply distinguish between 8087, 80287, and 80387. This also applies to the 80487, which is simply a disguised 80486, in which the 80486 part has been switched off.

Differentiating between 8087 and 80287/80387

A flag, called Interrupt-Enable-Mask, in the control word differentiates the 8087 from the 80287 and 80387. In the 8087, this flag decides whether the coprocessor should trigger an interrupt upon receiving an invalid machine language command. Since

Differentiating between 8087 and 80287/80387

A flag, called Interrupt-Enable-Mask, in the control word differentiates the 8087 from the 80287 and 80387. In the 8087, this flag decides whether the coprocessor should trigger an interrupt upon receiving an invalid machine language command. Since the 80287 and 80387 use exceptions instead of interrupts, this flag is always set to zero on 80287/80387s and cannot be set to 1. You can check this by first loading the control word to set the flag to 0 and then using the FDISI machine language command to set the flag to 1. On an 80287/80387, this command won't have any effect. You can also confirm this by reading the control word and checking this flag.

Differentiating between an 80287 and 80387

If you don't have an 8087 coprocessor, this process becomes more complicated because flags can't be used to distinguish between them. However, the 80287 and 80387 processors differ in their ability to process infinite values, called infinities. To create an infinity within the test routine, divide one by zero. The result of this operation is duplicated, its sign reversed, and then two values are compared. Since switching the sign of infinity is possible only on the 80387, the two values will be different only on an 80387 coprocessor. Verify this by using the appropriate commands and then establish the identity of the coprocessor.

Sample programs

You can see how this theory works by reading the processor and its helper in two assembler modules, which were designed for linking to Pascal and C programs. These modules are called PROCPA.ASM and PROCCA.ASM; both of them contain two routines named GETPROC and GETCO. These routines represent functions that return a numeric value which characterizes the type of processor (GETPROC) or the type of coprocessor (GETCO).

The high level language programs PROCP.PAS and PROCC.C clearly illustrate this by using the return values of these functions as indexes in string arrays. The names of the various Intel processors and coprocessors are stored in these string arrays.

Accessing the 32 bit register

When you examine the assembler modules, which are identical, you'll notice the numerous DB commands in the GETPROC function. These commands represent the machine codes of the various commands used to distinguish between 80386 and 80486. This is where machine language commands of the 80386 and 80486, which rely on the extended 32 bit register of these processors (EAX, EBX etc.), are used. However, these commands can be used only if the assembler is informed, by the pseudo command .386, these commands are allowed.

Unfortunately this has an unpleasant side effect. The assembler gives the attribute "32-Bit" to the code segment and records this in the object file. The linker, which links the assembler module with the high level language model, doesn't approve of this because the various segments are no longer compatible. To avoid this conflict, the commands are specified directly in machine code so the assembler doesn't recognize them as enhanced 386 commands.

You'll find the following program(s) on the companion CD-ROM



PROCP.PAS (Pascal listing)
PROCPA.ASM (Assembler listing)
PROCC.C (C listing)
PROCCA.ASM (Assembler listing)



DOS File System

The user doesn't see many of the tasks that DOS performs. This is why some users underestimate the complexity of DOS. For example, DOS requires many data structures for handling a mass storage device, although this is not always realized by the user. The file system must perform many steps before executing even small tasks, such as copying files or searching for a file on the hard drive.

Disk drives recognize tracks, sectors, and read/write heads instead of files and subdirectories. Therefore, DOS's file system addresses all available mass storage devices at both physical and logical levels. A file system of this type consists of a series of data structures, which describe the capacity and contents of a disk drive. In this chapter we'll discuss how the DOS file system is designed and how it works.

Basic Structure Of The File System

The volume is the basis of the DOS file system. From the user's viewpoint, DOS addresses mass storage devices as volumes. Each individual volume is assigned a letter. Floppy disk drives are identified by the letters A and B, while the letters C or D usually identify a hard drive. Although a hard drive can be divided into multiple volumes, a floppy disk drive can consist of only one volume.

DOS Versions 3.3 and lower limited volume size to a maximum of 32 megabytes. Therefore, any hard drives with capacities over 32 megabytes were divided into multiple volumes. In this chapter we explain how DOS Version 4.0 broke the 32 megabyte barrier.

Every DOS volume has its own structure, regardless of whether it's intended for diskettes or hard media. The size of the storage media isn't relevant to the structure because it only affects the number of individual data structures required to manage the volume.

Volume label names

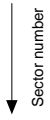
Although not required, each volume can be assigned a volume label name when created. The DIR command lists volume label names when they're available. Each volume has its own root directory, which can contain multiple subdirectories and files. These subdirectories and files can be maintained and manipulated by using one or more of the interrupt 21H functions.

Sectors

DOS subdivides each volume into a series of sectors organized sequentially. Each sector contains a specific number of bytes (usually 512) and is assigned a consecutive logical sector number, beginning with sector 0. A 10 megabyte volume contains 20,480 sectors, consisting of logical sectors 0 to 20,479. DOS cannot control the physical arrangement of the sectors. This is controlled by the device driver, which mediates between the volume and DOS. The device driver's distribution of logical sectors (organized or not) is unimportant. It's only important the device driver clearly differentiates between logical and physical sectors.

Since DOS API function calls with interrupt 21H are directed to files instead of individual sectors, DOS converts these file accesses into sector accesses. To do this, DOS uses the volume's directory structure and a data structure known as the FAT (File Allocation Table).

The following table shows the basic structure of a mass storage device:

Mass storage device structure		
	0	Manufacturer's name, device driver, boot routine First file allocation table (FAT) One or more copies of FAT Root directory with volume label names Data register for files and subdirectories

As we mentioned, every volume is divided into areas containing the various DOS data structures and individual files. The FORMAT command creates these data structures when you format a disk. Since the size of the individual areas can differ depending on the type of mass storage device (and the manufacturer), every volume contains a boot sector.

The Boot Sector

The boot sector contains all the information required to access the different areas and data structures. DOS creates this sector during disk formatting. Boot sectors always have the same structure and are always located in sector 0 so DOS can find and interpret it properly. The table on the right lists the layout of the boot sector.

The term "boot sector" is used because DOS "boots" (i.e., starts) from this sector. Since DOS usually isn't stored in permanent PC memory (ROM), it's loaded and started from disk. After you switch on the computer, the BIOS takes over system initialization. It loads physical sector 0 (not logical sector 0) of the floppy disk or hard drive into memory. Since device drivers aren't in memory, the BIOS checks physical sector 0 for information. It then loads the boot information from that sector. Once it completes its work, the BIOS starts execution at address 0.

The boot sector always contains an assembly language JUMP instruction at address 00H. Execution of the boot sector's program code begins at this address. After execution, the program continues at a location further into the boot sector. This instruction can be either a normal jump instruction or a "short jump." The field for this jump instruction is 3 bytes long, but a "short jump" only requires 2 bytes. Therefore, a NOP (No Operation) instruction always follows the "short jump" to fill in the extra byte. As its name suggests, this NOP instruction doesn't do anything.

A series of fields, which contain certain information about the organization of the media, follow. The first field is 8 bytes long and contains the manufacturer's name, where the medium was formatted, as well as the DOS version number that performed the formatting. The field may also contain the name of a software manufacturer (e.g., if a program such as PCTools formatted the volume).

The next fields contain the physical format of the media (i.e., the number of bytes per sector, the number of sectors per track, etc.) and the size of the DOS data structures stored on the medium. The physical device information is needed when using the interrupt 13H BIOS functions. These fields

Boot sector layout		
Address	Contents	Type
00H	Jump to boot routine (E9xxx or EBxx90)	3 bytes
03H	Manufacturer's name and version number	8 bytes
0BH	Bytes per sector	1 word
0DH	Sectors per cluster	1 byte
0EH	Number of reserved sectors	1 word
10H	Number of FATs	1 byte
11H	Number of entries in root directory	1 word
13H	Number of sectors in volume	1 word
15H	Media descriptor	1 byte
16H	Number of sectors per FAT	1 word
18H	Sectors per track	1 word
1AH	Number of read/write heads	1 word
1CH	Number of hidden sectors	1 word
1EH-1FFH	BOOT ROUTINE	

NOTE

Some sources of undocumented DOS structures state the BPB is a parameter table, which can be accessed by using Get Service Data, instead of part of the boot sector. These sources imply the boot sector information is only part of the BPB.

are called the BIOS parameter block (BPB). DOS uses this information for various tasks.

Three additional fields, providing other volume information to the device driver, follow the BPB. However, these fields aren't used directly by DOS.

Bootstrap

Next is the bootstrap routine, to which the jump instruction branches at the beginning of this boot sector. This routine handles the loading and starting of DOS through the individual system components (see Chapter 3).

Several reserved sectors may follow the boot sector. These sectors can contain additional bootstrap code. The numbers of these sectors are recorded in the BPB in the field starting at offset address 0EH. This field terminates the boot sector; a 1 in this field indicates that additional reserved sectors don't follow the boot sector. This applies to most PCs, because no versions of DOS have required a bootstrap loader that cannot fit into the first boot sector.

The File Allocation Table (FAT)

DOS must know which sectors of the volume are still available before it can add new files or enlarge existing files. This information is contained in a data structure called the FAT (File Allocation Table), located immediately next to the media's reserved area. Each entry in the FAT corresponds to a certain number of logically contiguous sectors called clusters, on the media. Location 0DH of the boot sector specifies the number of sectors per cluster as part of the BIOS Parameter Block. Only powers of 2 (1, 2, 4, 8, etc.) are acceptable values. On an XT hard drive, this location contains the value 8 (8 consecutive sectors form a cluster). AT, 386, and 486 hard drives have only 4 sectors per cluster.

As the table on the right shows, the number of sectors comprising a cluster depends on the storage medium.

Sectors per cluster	
Device	Sectors per cluster
Single sided disk drive	1
Double sided disk drive	2
AT hard drive	4
XT hard drive	8

Despite the values in the previous table, a formatting utility, such as FORMAT, isn't limited to these cluster values. This applies particularly to volumes containing more than 32 megabytes. DOS Versions 4.0 and higher accommodate larger volumes by adding clusters. A volume larger than 32 megabytes quickly exceeds the 65,536 limit for clusters. So, the capacity of the file system can be extended at will, without changing the current structure, by including more sectors per cluster.

The following table (lower right) shows the clustering used by DOS Version 4.0 for volumes larger than 32 megabytes and up to 2 Gigabytes.

This clustering allocates more memory to the last cluster of a file. This cluster is filled only when its size represents a multiple of the cluster size.

Volume and Cluster sizes of DOS 4.0					
Volume size (megabytes)	128	256	8K	16K	32K
Cluster size	2K	4K	8K	16K	32K
Sectors per cluster	4	8	16	32	64

File fragmentation

The idea of joining several sectors into a cluster is based on the logic used by DOS to write files to a medium. Instead of selecting adjoining sectors for file storage, DOS fragments (disassembles) the file so the various pieces can fit into the available sectors.

The following explains the reasons why DOS fragments the file. Users are constantly creating and deleting files. If you start with an empty volume, new files and their respective clusters are stored in sequence. However, when you delete a file, a gap may develop between two sectors. DOS doesn't adjust the clusters on a volume because this process requires too much time. This is especially true for larger volumes. So, DOS places new files in the gaps created when the older files were deleted. If the entire file could be stored in one of these gaps, it could be kept in one contiguous unit. However, this usually doesn't happen. As a result, DOS divides files and fits them into available gaps.

This process slows file access because the read/write head must be repositioned after almost every read function. To avoid an excessive disassembly of the file, DOS gathers several sequential sectors on the media into a cluster. This ensures that at least the sectors of a cluster contain a portion of a file. If DOS didn't use clusters, a file of 24 sectors could be stored in numerous sectors, which would require the read/write head to be positioned a maximum of 24 times to read the entire file. The cluster principle saves a lot of time, since a file comprised of 4 sectors per cluster is stored in 6 clusters and the read/write head must be repositioned only 6 times.

However, there is a problem with this process. Since a file is assigned at least one cluster, some storage space is wasted. Consider the AUTOEXEC.BAT file, which is usually no longer than 150 bytes. Normally, this file could be stored on a single sector (and still waste almost 400 bytes). However, AUTOEXEC.BAT occupies a cluster of 2048 bytes on an AT, which wastes more than 1.5K of hard drive space.

Disk optimizing programs, such as BeckerTools Disk Optimizer or DEFRAG in MS-DOS 6.x, solve this problem by reorganizing the medium and storing all the files in consecutive clusters.

The FAT layout

Now let's return to the file allocation table. The size of individual entries in the FAT under DOS Versions 1.0 and 2.0 is 12 bits. For DOS Versions 3.0 and up, the size of an entry in the FAT depends on the number of clusters. If a volume has more than 4,096 clusters, then each FAT entry is 16 bits; otherwise each FAT entry is 12 bits.

A 12-bit FAT permits control of 4,096 clusters, which corresponds to 4 sectors per cluster, providing a total of 8 megabytes. Although this amount could be expanded by adding more sectors to a cluster, such an expansion isn't recommended. Therefore, you'll find only 16-bit FATs on newer hard drives of 20 megabytes and up, thus allowing the 65,536 maximum of addressable clusters.

The number of bits per FAT entry must be determined before file access. The information in the BIOS parameter block is used for this purpose. The total number of sectors in the volume can be found starting at location 13H. Divide this number by the number of sectors per cluster to obtain the number of clusters in the volume.

The first two entries of the FAT are reserved and aren't related to the cluster assignment. Depending on the sizes of the individual entries, 24 bits (3 bytes) or 32 bits (4 bytes) can be available. The first byte contains the media descriptor, while the value 255 fills in the other bytes. The media descriptor, which is also stored in address 15H of the BPB, indicates the device the media uses (e.g., a diskette). The table on the right shows which codes are possible.

Perhaps you're wondering why the individual entries of the FAT are 12 or 16 bits wide if all they do is show whether a cluster is occupied. This could have been done with one bit; the bit could contain 1 when the cluster is occupied and 0 if the cluster is available.

The entries in the FAT help mark the available clusters and identify the individual clusters containing a specific file. The directory entry of a file tells DOS which cluster holds the first sectors of a file. The number of this cluster corresponds to the number of the FAT entry belonging to it. In this entry is the number of the cluster containing the next sector of file data. This search continues until the last cluster of a file has been reached.

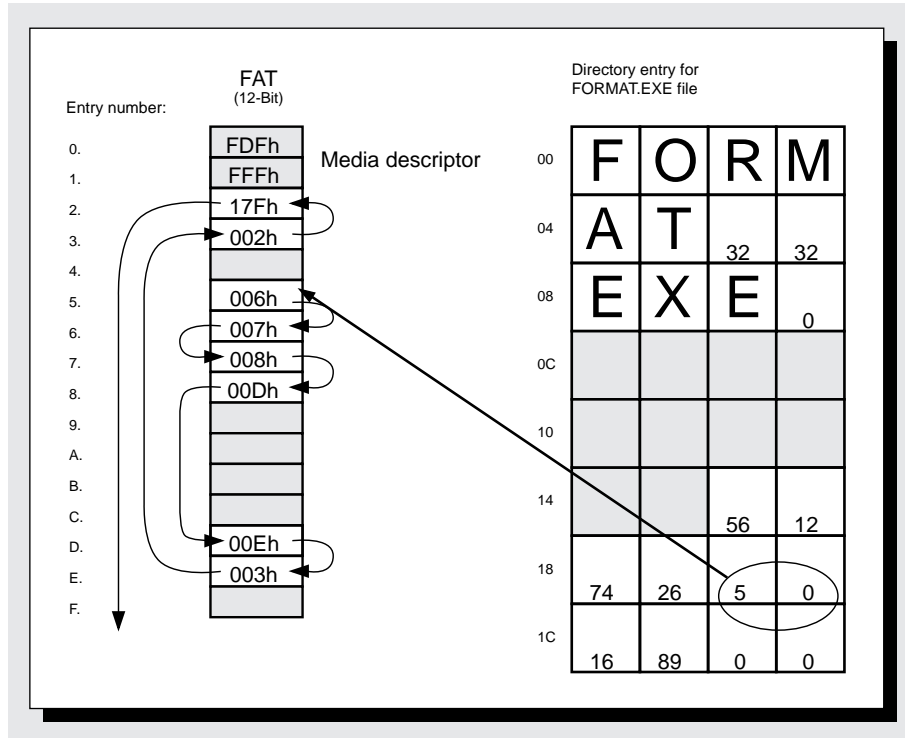
However, the cluster numbers refer to the beginning of the data structure rather than the beginning of the volume (more on this later). For example, assume the beginning of a file resides in the fourth cluster of the volume. You must read the fourth

Media descriptor codes		
Code	Device	Description
F0H	3.5" disk drive	2 sides, 80 tracks, 18 sectors per track
F8H	Hard drive	Varies
F9H	5.25" disk drive	2 sides, 80 tracks, 15 sectors per track
	3.5" disk drive	2 sides, 80 tracks, 9 sectors per track
FAH	5.25" disk drive	1 side, 80 tracks, 8 sectors per track
	3.5" disk drive	1 side, 80 tracks, 8 sectors per track
FBH	5.25" disk drive	2 sides, 80 tracks, 8 sectors per track
	3.5" disk drive	2 sides, 80 tracks, 8 sectors per track
FCH	5.25" disk drive	1 side, 40 tracks, 9 sectors per track
FDH	5.25" disk drive	2 sides, 40 tracks, 9 sectors per track
FEH	5.25" disk drive	1 side, 40 tracks, 8 sectors per track
FFH	5.25" disk drive	2 sides, 40 tracks, 8 sectors per track

entry of the FAT to obtain the number of the next cluster containing the file, remembering that a FAT entry can be either 12-bit or 16-bit. This fourth entry contains the number of the next file cluster.

As the following illustration shows, a chain is formed in which the individual clusters assigned to a file can be located in the proper sequence. The logical number of each sector can be derived from the cluster number. DOS must send this logical number to the device driver before these sectors can be read or written by DOS. DOS device drivers operate at sector level rather than cluster level. To convert clusters to logical sectors, multiply the cluster number by the number of sectors per cluster.

FAT entry and file clusters



The FAT entry corresponding to the last cluster of a file must contain a special code that tells DOS the file ends here. A cluster number indicates this code, which is either greater than FF8H (12-bit) or greater than FFF8H (16-bit). However, the following tables show the meanings of the various FAT entries. For example, cluster numbers FF0H-FF6H (12-bit) or FFF0H-FFF6H (16-bit) indicate a reserved cluster (i.e., the root directory of a volume). The root directory has a fixed position and size, and the cluster can be independent rather than chained to the FAT.

Cluster codes FF7H (12-bit) and FFF7H (16-bit) also have special significance. These codes identify clusters whose sectors contain errors, ensuring that DOS doesn't write data to these bad clusters.

Cluster code 0H indicates unoccupied clusters, as well as the first cluster of the volume. Like the first cluster, code 0H represents no sectors, because it contains the volume's medium type. As you'll see later in this book, this is why the cluster number must be reduced by two when converting cluster numbers into sector numbers.

12-bit FAT cluster codes		16-bit FAT cluster codes	
Code	Meaning	Code	Meaning
000H	Cluster is available	0000H	Cluster is available
FF0H-FF6H	reserved cluster	FFF0H-FFF6H	Reserved cluster
FF7H	Cluster damaged, not used	FFF7H	Cluster damaged, not used
FF8H-FFFH	Last file cluster	FFF8H-FFFFH	Last file cluster
xxxH	Next file cluster	XXXXH	Next file cluster

Multiple copies of the FAT

The FAT's importance in the DOS file system becomes obvious when a virus or hardware error damages the FAT. All files and subdirectories are still available in principle. However, they remain inaccessible to the user because DOS can no longer gather the individual clusters into one unit.

DOS is designed so several identical copies of the FAT can be kept on a volume. The boot sector's offset address 10H contains the number of FATs. If DOS finds a medium of this type, it automatically updates all copies of the FAT and records this update in offset 10H while creating or deleting files. So, if one FAT is damaged, it can be replaced with another, which minimizes data loss.

The DOS CHKDSK command tests the various FATs to see if they are identical. If the primary FAT is damaged, CHKDSK replaces the damaged primary FAT with another FAT.

The Root Directory

Now let's look at the structure of a directory. The root directory of a volume immediately follows the last copy of the FAT. This root directory (like all subdirectories) consists of 32-byte entries, in which information about individual files, subdirectories, and volume label names can be stored. The maximum number of entries in the root directory, and therefore its size, is stored in the BPB starting at address 11H. The FORMAT command specifies both the size number and the BPB. Before considering individual fields of this data structure, the table on the right provides an overview of a directory entry.

Directory entry layout		
Address	Contents	Type
00H	Filename (blanks padded with spaces)	8 bytes
08H	File extension (blanks padded with spaces)	3 bytes
0BH	File attribute	1 byte
0CH	Reserved	10 bytes
16H	Time of last change	1 word
18H	Date of last change	1 word
1AH	First cluster of file	1 word
1CH	File size	2 words

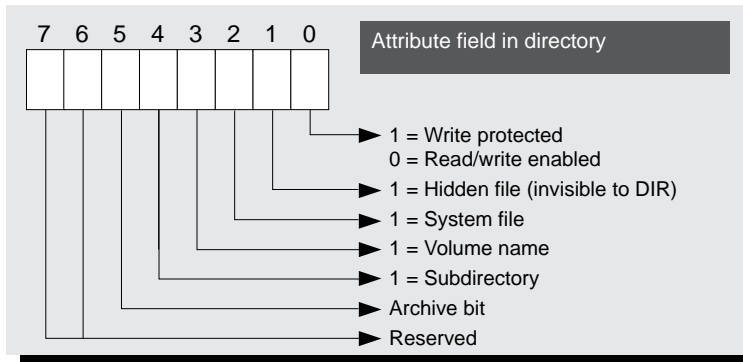
The first eight bytes usually contain the name of the current file. If the filename is shorter than eight characters, DOS fills the remaining characters with spaces (ASCII code 32).

If the directory entry doesn't contain information about a file, but the file is used in another way, the first byte of the filename (therefore the first byte of the directory entry) is identified by a special code (see the table on the right).

The first byte of the directory entry	
Code	Meaning
00H	Last directory entry
05H	First character of filename has ASCII code E5H
2EH	File applies to current directory
E5H	File deleted

The second field contains the three character filename extension. If the extension is less than three characters long, DOS fills in the extra characters with blank spaces (ASCII code 32). The period between filename and extension is displayed by the DOS command DIR but isn't kept in the directory; DIR displays this character so the names are easier to read.

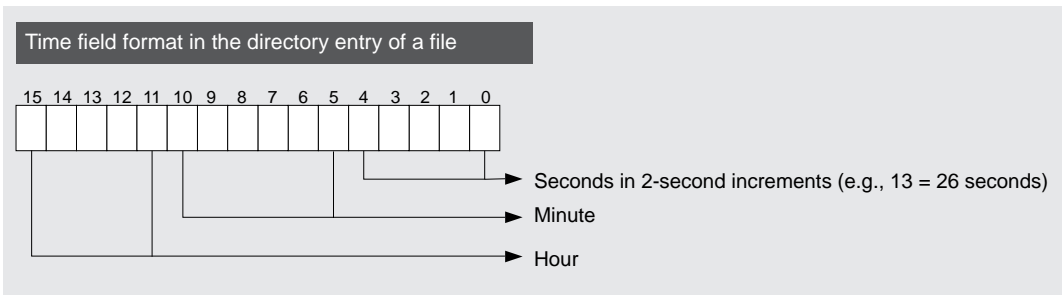
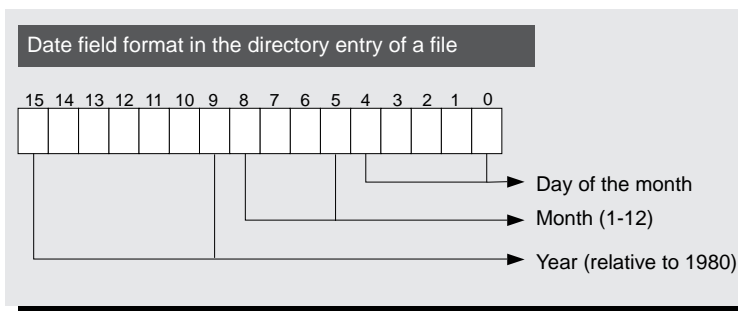
The one-byte attribute field is next. As shown in the following figure, the individual bits of this field define certain attributes. The various attributes can be combined so a file (as in the IBMBIOS.COM file) can have the attributes READ_ONLY, SYSTEM, and HIDDEN.



A reserved field follows the attribute field. DOS uses this field for internal operations, and some sources claim that Novell NetWare uses this bit for sharing data.

While the significance of bits 0 to 4 is easy to see, the significance of bit 5 needs additional explanation. The name archive bit comes from its use in making backup copies. Every time a file is created or modified, this bit is set to 1. If a program is used to backup this file, (for example the DOS BACKUP command), the archive bit is reset to 0. The next time the BACKUP command is used, it can determine, from the archive bit, whether this file has been modified since the last backup. If it still contains the value 0, the file doesn't have to be backed up again. If the archive bit contains a 1, the file was modified and should be backed up again.

The attributes volume label name and subdirectory will be discussed in more detail later. A reserved field, which DOS requires for internal operations, follows the attribute field. The time and date fields indicate when the file was last created or modified. Both are stored as words (2 bytes), but have special and different formats.



The next field shows the number of the cluster that contains the first data of the file. It also shows the number of the FAT containing the number of the next cluster assigned to the file. This field forms the beginning of a chain through which all the clusters assigned to a file can be retrieved.

The file size in bytes is stored in two words with the lower word stored first. Using a small formula and the two words, the file size can be calculated as follows:

$$\text{File size} = \text{word1} + \text{word2} * 65,536$$

Subdirectories

Both subdirectories and volume label names deserve special consideration. The volume label name can exist only in the root directory. It's indicated by bit 3 of the current directory entry's attribute field. The filename in a volume entry acts as the volume label name. Use the DOS commands DIR, VOL, and TREE to display the volume label name.

If bit 4 of the current directory's attribute field is set, then this entry is for a subdirectory. If bit 1 in this field is also set, the subdirectory can be addressed. However, this subdirectory isn't displayed when you execute the DIR command. For these entries, the filename and extension field contain the subdirectory name; the date and time field contain the time of its creation. The file length field is always 0. The field that usually indicates the first cluster of the file now indicates the cluster that contains the directory entries of this subdirectory. They have the same 32-byte structure as the entries in the root directory.

As in a normal file, the entry in the FAT corresponding to the subdirectory cluster points to the next cluster of the subdirectory. This is true as long as one cluster is enough for the directory of the subdirectory. This doesn't apply to the root directory, which extends through several sectors or clusters that follow each other logically. Also, the individual clusters of the root directory cannot be connected through the FAT, because the FAT only refers to the data area of the volume. This is the area that accepts files and subdirectories, but not the root directory.

The process previously described reveals that DOS separates the individual files in a storage unit according to their directories. Instead of storing the files of one directory in one area, DOS scatters the files across the storage medium.

When a subdirectory is created, two files are created with the names '.' and '..'. These files can be erased only when you remove the entire subdirectory. The first file points to the current subdirectory. Its cluster field contains the number of the first cluster of the current subdirectory. The second entry points to the parent directory located before the current directory in the directory tree. If the parent directory is the root directory, the cluster field contains the value 0. The path to the root directory can be traced through this entry, since, as every subdirectory searches for its parent directory, it comes closer to the root directory.

Now let's return to our discussion of mass storage device structures. The file area follows the root directory. This area, which occupies the remaining storage area of the mass storage device, accepts the individual files and various subdirectories. There is an entry in the FAT corresponding to every cluster in this area. If a file is enlarged, DOS reserves a cluster that is still available to store the additional data of the file. The FAT entry of the last cluster, which previously indicated the end of the file, is changed to point to the new cluster. This in turn contains the new end character.

Both DOS Versions 1.0 and 2.0 search for unused clusters from the beginning. In DOS Versions 3.0 and higher, a more complicated search procedure is used to try to select an unused cluster near the other clusters comprising the file. This reduces the access time to the file. Conversely, when reducing file size or deleting a file, the FAT is updated to indicate the unused clusters are again available. They can be used again when a new file is created or expanded.

Let's begin at the point where DOS finds the first cluster or a file and the FAT entry for the next cluster in the first cluster. We need to calculate the sector from this cluster.

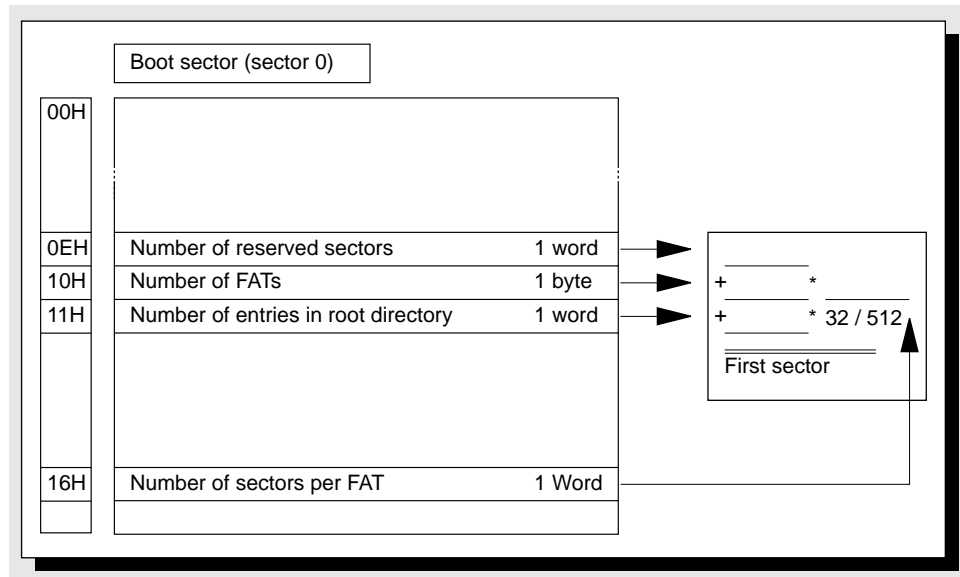
1. Subtract 2 from the cluster number. The first two FAT entries contain the media descriptor, so FAT entry 2 is the actual zero cluster on the volume.
2. Multiply the cluster by the number of sectors per cluster. BIOS obtains this information from the BIOS Parameter Block loaded from the boot sector. The result relates to the first sector of the volume, instead of the first sector of the

data range (which is the result we want). You can compute the logical sector where the data range starts, also using information from the BIOS Parameter Block.

3. The boot sector, the FAT and its duplicates, and the root directory precede the volume's data range. The lengths of these ranges must be calculated and added together. Read the number of sectors reserved from offset address 0EH of the boot sector. Read the number of FAT sectors from offset address 16H of the boot sector, and multiply the number of FAT sectors by the number of FATs (found in address 10H), then add the total to the number of reserved sectors.
4. DOS also requires the number of sectors occupied by the root directory. This number is stored in the word beginning at offset 11H in the boot sector. Multiply this value by 32 (bytes per entry), and divide the result by 512 (bytes per sector). Add the result to the boot sector and FAT lengths to obtain the number of the first sector in the data range.

DOS performs this calculation only when booting, or when a medium has been changed (e.g., every time you change diskettes in a floppy disk drive).

Calculating the first sector of the data range



By adding the number of the first sector in the data range to the first sector number of the addressed file, you receive the logical sector number. DOS can then pass this logical sector number to the device driver for file access.

When DOS requires access to both the first cluster and subsequent clusters of the file, it must read those subsequent clusters from the FAT. These calculations vary depending on the widths of the FAT entries. Reading the next adjacent cluster is easy to do with a 16-bit FAT. Simply multiply the cluster number by 2 to find the next cluster. The result represents the offset address relative to the beginning of the FAT in memory, where the next cluster can be found.

If DOS is dealing with a 12-bit FAT, multiply the cluster number by 1.5. The whole number part of the product becomes the offset in the FAT. The word at this memory address is read. If the product is a whole number, the word read must be combined with a logical AND to 0FFFH to obtain the number of the next cluster. If the product isn't a whole number, the AND is omitted and the word is shifted by 4 bits to the right (divided by 16).

DOS usually doesn't have to load the FAT into memory, because it permanently stores a copy of this data structure in memory to save execution time.

Now DOS knows the next cluster of the file. If you're using a 12-bit FAT, the next cluster lies in the range from FF8H to FFFH. If you're using a 16-bit FAT, the next cluster lies in the range from FFF8H to FFFFH. This process repeats until DOS finds the end of the file.



Internal Structure Of DOS

In this chapter we'll discuss the internal structure of DOS and the booting process. Since these two items occur in everyday DOS programming, the programmer should understand what happens behind the scenes.

Components Of DOS

Several major components comprise DOS, each with a certain task within the system. The three most important components are the DOS-BIOS, the DOS kernel, and the command processor. Each appear in a separate file.

DOS-BIOS

DOS-BIOS is stored in a system file that appears under various names (IBMBIO.COM, IBMIO.SYS or IO.SYS). This file has the file attributes Hidden and Sys, which indicates that this system file doesn't appear when the DIR command is entered. The DOS-BIOS contains the device drivers for the following units:

```
CON      (Keyboard and Display)
PRN      (Printer)
AUX      (Serial Interface)
CLOCK    (Clock)
Disk drives and/or hard disks which have the drive specifiers A, B and C
```

If DOS wants to communicate with one of these, it accesses a device driver contained in this module, which in turn uses the routines of ROM-BIOS. The DOS-BIOS (i.e., the connection between individual device drivers and other hardware dependent routines) are the most hardware dependent components of the operating system and vary from one computer to another.

Don't confuse the device drivers in this module with the installable device drivers. The DOS-BIOS device drivers cannot be changed by the user.

DOS kernel

The DOS kernel in the IBMDOS.COM or MSDOS.SYS file is usually invisible to the user. It contains file access routine handles, character input and output, and more; it immediately follows the file IBMIO.SYS or IO.SYS. Both sets of files are assigned the SYSTEM, HIDDEN, and READ-ONLY file attributes. These attributes indicate that these files directly affect the system. So you can't view them or delete them by normal means.

These files contain the various DOS-API functions, which are called using interrupt 21H. The routines operate independent of the hardware and use the device drivers of DOS-BIOS for keyboard, screen, and disk access. The module can be used by different PCs without being limited to one machine. User programs can access these functions in the same way as the ROM-BIOS functions every function can be called with a software interrupt. The processor registers pass the function number and the parameters.

Command processor

Unlike the two modules we've described, the command processor is contained in the file named COMMAND.COM. It displays the "A>" or "C>" prompt on the screen, accepts user input, and controls input execution. Many users incorrectly think

that the command processor is actually the operating system. Actually it's only a special program that executes under DOS control.

The command processor, also called a shell in programmer's terminology, actually consists of three modules: A resident portion, a transient portion, and the initialization routine.

The resident portion (the part that's always in the computer's memory) contains various routines called critical error handlers. These allow the computer to react to different events, such as pressing the **Ctrl****C** or **Ctrl****Break** keys or errors during communication with external devices (e.g., disk drives and printers). The latter causes the message:

```
Abort, Retry, Ignore
```

or

```
Abort, Retry, Fail
```

The transient portion contains code for displaying the (A>) prompt, reading user input from the keyboard, and executing the input. The name of this module is derived from the fact that the RAM memory where it's located is unprotected and can be overwritten under certain circumstances. When a program ends, control returns to the resident portion of the command processor. It executes a checksum program to determine whether the transient portion was overwritten by the application program. If it was, the resident portion reloads the transient portion.

The initialization portion loads during the booting process and initializes DOS. This part of the command processor will be examined in detail in the next chapter. When its job ends, it's no longer needed and the RAM memory it occupies can be overwritten by another program. The commands accepted by the transient portion of the command processor can be divided into three groups: internal commands, external commands and batch files.

Internal commands lie in the resident portion of the command processor. COPY, RENAME, and DIR are internal commands.

External commands must be loaded into memory from a diskette or hard disk as needed. FORMAT and CHKDSK are external commands.

After execution the command processor releases the memory used by these programs. This memory can then be used for other purposes.

Batch files

A batch file is a text file containing a series of DOS commands. When a batch file is started, a special interpreter in the transient portion of the command processor executes the batch file commands. Execution of batch file commands is the same as if the user entered them from the keyboard. An important batch file is the AUTOEXEC.BAT file, which executes immediately after DOS is first loaded.

Like all commands of a batch file, these commands are checked for internal commands, external commands, or calls to other batch files. If the first is true, the command executes immediately, since the code is already in memory (in the transient part of the command processor). If it's an external command or another batch file, the system searches the current directory for the command. If such a file doesn't exist in this directory, all directories specified in the PATH command are searched in sequence. During the search, only files with the .COM, .EXE, or .BAT extensions are examined.

Since the command processor cannot search for all three extensions simultaneously, it first searches for files with .COM extensions, then for .EXE files, and finally for .BAT files. If the search is unsuccessful, the screen displays an error message and the system waits for new input.

18. Internal Structure Of DOS

Booting DOS

The interaction between the DOS-BIOS, the kernel, and the command processor is most obvious when a program is in memory. However, the process of booting the system also calls these modules.

Searching for boot files

When a PC is switched on, the program contained in ROM begins executing. This ROM program is sometimes called the ROM-BIOS, POST (power on self test), resident diagnostics, or bootstrap ROM. It performs several tests on the hardware and memory and then starts to load the DOS.

First, the PC checks for a disk in the floppy disk drive. If a disk exists in the floppy disk drive, the PC checks the disk for the boot sector. If a disk isn't in the drive, the PC searches for a hard disk from which to boot DOS. If a hard disk doesn't exist, the PC displays an error message asking the user to insert a system disk.

The first sector on a bootable floppy disk or hard disk is called the boot sector. The program in the boot sector is read into memory and executes. First it checks for the presence of two files: IBMBIO.COM (sometimes called IO.SYS) and IBMDOS.COM (sometimes called MSDOS.SYS). A bootable floppy disk or hard disk must contain these two files or an error message is displayed. Next these program files are loaded into memory.

The program file IBMBIO.COM consists of two modules. The first contains the basic device drivers-keyboard, display, and disk. The second contains the initialization sequence for DOS. When the IBMBIO.COM program executes, it continues to initialize the system by moving the DOS kernel (loaded in the IBMDOS.COM program file) to the last available memory location.

The DOS kernel builds several important tables and data areas, and performs initialization procedures for individual device drivers that were loaded with the IBMBIO.COM program file.

Next, DOS searches the boot disk for a file named CONFIG.SYS. If this file is found, the commands contained in the file are executed. These commands add device drivers to DOS, allocate disk buffers and file control blocks for DOS, and initialize the standard input and output devices.

Finally, the command processor COMMAND.COM (or other shell specified in the CONFIG.SYS file) is loaded and control is passed to it. The booting process ends and the initialization routines remain as "garbage" data in memory until overwritten by another program.



COM And EXE Programs

Along with batch files, DOS also recognizes program files that have the .COM and .EXE file extensions. These extensions indicate these files have different properties and they are executable. These program types differ from others because of the way their program code is stored and the maximum program sizes that are allowed.

The differences between these program types aren't important to a programmer working in high level languages. The programmer only needs to know the program runs; the file format doesn't matter. Also, development packages, such as Turbo Pascal and QuickBASIC, create only EXE files. Some C compilers can produce COM programs using the TINY memory model, in which the combination of program code, data segment, and stack occupies 64K of RAM or less.

The only advantage COM programs have over EXE programs is their slightly smaller program size, which is often only a few hundred bytes.

The programmer working in a high level language doesn't have to worry about the formats and peculiarities of COM and EXE programs because the compiler handles this. However, this information is important to the programmer that's developing software in assembly language.

In this chapter we'll describe the structure and functions of these two program types.

Differences Between COM And EXE Programs

The COM program is basically a relic from the CP/M era, when RAM was minimal and programs weren't larger than 64K. In DOS, a COM program cannot be larger than 64K. An EXE program can be as large as the memory capacity available to DOS or even larger. (Instead of being loaded into memory, portions of the EXE program may be reserved for later use.)

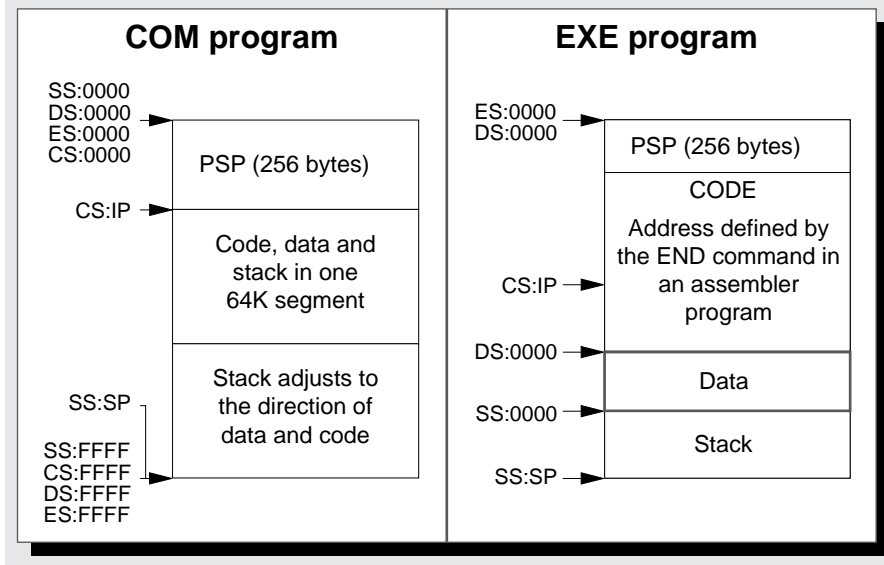
In a COM program, the program code, data, and stack are stored in one 64K partition. All of the segment registers are set at the start of the program and remain fixed for the duration of the program execution. They point to the start of the 64K memory segment. However, the contents of the ES register may be changed because that register doesn't directly affect program execution.

Except for the ES registers, these values must also be stored during program execution. Program code in this segment can be addressed through the CS register, the data can be addressed through the DS register, and the stack can be addressed through the SS register. However, there are exceptions to the rule. For example, if you call a DOS function, which expects another segment address in the DS register, the DS register may be loaded with another value before the function call. Neither DOS nor the processor will object to this. After execution, the DS register must be reloaded with the COM program's segment address to make the global variables of the program or other data accessible.

Unlike COM files, a direct sequence doesn't exist for EXE file segments. Code, data, and the stack are stored in different segments according to size; they could be distributed over several segments. This distribution applies to larger, commercial programs (e.g., Microsoft Word), which contain program code consisting of several hundred kilobytes. So, during the execution of an EXE program, the various segment registers point to individual code, data, or stack segments instead of a general memory segment.

Regardless of the program type, DOS creates a data structure in memory called the Program Segment Prefix (PSP) before the program starts. (We'll discuss this in more detail later.) This data structure contains 256 bytes. It immediately precedes the program in memory, as shown in the following illustration:

*Comparing
COM and EXE
programs in
memory*



COM Programs

COM program files are stored on disk as an image copy of memory. Because of this, no further processing is needed during loading. So, COM programs load and start execution faster than EXE programs. Also, COM programs are usually more compact than EXE programs because other information, besides the program code and the initialized variables, are also stored there.

Although these items may have been important in the past, they are no longer significant. Programs have become so large that two or three hundred additional bytes no longer make that much difference in an EXE file. Also, computers have become so fast the speed advantage to loading a COM program has been reduced to a few milliseconds. Only a few dedicated assembly language programmers still use COM format when a program must be as compact as possible.

Microsoft and COM programs

Microsoft's COM program support is steadily decreasing. The Microsoft Assembler MASM assembles files as EXE programs. To convert these programs to COM programs, you need EXE2BIN, which is an important utility packaged with older versions of MS-DOS. MS-DOS 4.0 didn't contain EXE2BIN. The version check in all DOS programs ensured that copies of EXE2BIN, which were taken from older versions of MS-DOS, would not function under MS-DOS 4.0 or 4.01.

So, programmers who wanted to continue using EXE2BIN were forced to alter DOS's version check in order to make EXE2BIN compatible with MS-DOS.

Unlike the LINK program included with the Microsoft Assembler MASM, the Turbo Assembler from Borland International can create both EXE and COM programs.

When MS-DOS Version 5.0 was released, Microsoft again packaged EXE2BIN with DOS.

Registers during start of program

A COM program loads immediately following the PSP. Execution then begins at the first memory location following the PSP at offset 100H. For this reason, a COM program must begin with an executable instruction, even if it's only a jump instruction to the actual start of the program.

COM program memory limits

As we mentioned, a COM program can be only 64K (65,536 bytes). The PSP (256 bytes) and at least 1 word (2 bytes) for the stack must also be reserved. Even though the length of the COM program can never exceed 64K, DOS reserves the entire available RAM for a program. So, DOS cannot allocate additional memory and the COM program cannot call another program using the EXEC function. You can bypass this limitation by releasing the unused memory for other uses with a DOS function.

When control is given to the COM program, all segment registers point to the beginning of the PSP. Because of this, the beginning of the COM program (relative to the beginning of the PSP) is always at address 100H. The stack pointer points to the end of the 64K memory segment containing the COM program (usually FFFEh). During every subroutine call within the COM program, the stack is adjusted by 2 bytes in the direction towards the end of the program. The programmer is responsible for preventing the stack from growing and overwriting the program, which would cause a crash.

There are several ways to end a COM program and return control to DOS or the calling program:

If the program runs under DOS Version 1.0, it can be terminated by calling interrupt 21H function 0 or by calling interrupt 20H. It can also be terminated by using the RET (RETurn) assembler instruction. When this instruction executes, the program continues at the address at the top of the stack. Since the EXEC function stored the value 0 at this location before turning control over to the COM program, program execution continues at location CS:0 (the start of the PSP). Remember that this location contains the call for interrupt 20H, which terminates the program.

Since the odds of a DOS Version 1.0 user are quite low, use DOS function 4CH. Microsoft recommends this function for terminating programs for all subsequent versions of DOS. The terminating program can pass a numeric return code to the calling program. For example, a value of 0 may indicate the program executed successfully, while a nonzero value indicates an error occurred during execution.

Developing COM programs in assembly language

Now we'll discuss the details the assembly language programmer must handle when developing a COM program. If you're a high level programmer, you may want to skip this section because you don't have to worry about these details. The compiler or interpreter handles them for you.

We've mentioned that COM programs are stored on a diskette or hard drive as a direct image of the machine code. Also, instead of being placed at a fixed address by DOS, COM programs can be placed at any memory location divisible by 16. This placement means that COM programs cannot contain FAR calls or specific segment addresses. Only NEAR calls, which contain offset addresses but no segment address, are allowed. Because of this, commands always refer to the current segment in the CS, DS, ES, or SS registers instead of to a certain segment address.

COM programs cannot contain assembly language instructions, such as LDS or LES. The assembler and linker accept these instructions, but EXE2BIN will refuse to make the assembled EXE file into a COM file. This also applies to TLINK, the Turbo Linker.

You may load constants. For example, you can load the segment address of video RAM into a segment register. This is possible because a reference isn't made to the segments of the program, whose position is uncertain, until the actual start of the program.

Assembly language instructions such as the following aren't allowed because the MOV instruction reads the segment address of the program:

```
MOV  AX, SEG PROGRAM
MOV  DS, AX
```

The following instructions are allowed because a constant segment address is used for reference:

```
MOV  AX, 0B000h
MOV  DS, AX
```

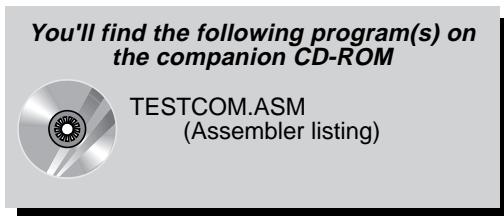
While the developer of a COM program is more limited in this instance than the developer of an EXE program, less work is required for the stack. Before the program is started, the stack is automatically placed at the end of the 64K COM segment by loading the SS register with the COM segment's address, and loading the SP register with the value 0FFFEH.

Before calling a COM program, DOS reserves all available memory for the program even though it normally uses only one 64K segment and indicates this by setting memory location 2 in the PSP. Usually the program terminates and the memory is made available to DOS again.

In some circumstances you may want to write a program that will remain resident after execution. However, DOS believes there isn't any memory available. This prevents other programs from loading and executing.

In other circumstances you may want to execute another program from this COM program using the EXEC function. Again, since DOS thinks that memory is unavailable, it won't allow the new program to run.

Both problems can be avoided by freeing up the unused memory.



Two ways are available to do this. Release only the memory outside of the 64K COM segment or release memory outside of the 64K COM segment and any unused memory within the 64K COM segment. Although this creates more memory for other programs, it relocates the stack outside the protected COM segment memory. So, the stack can be overwritten by other programs. Because of this, the stack must be relocated to the end of the code segment before releasing the memory. Also, the size of the stack must be limited (usually 512 bytes is sufficient).

The TESTCOM.ASM sample program on the companion CD-ROM demonstrates how to develop a COM program. A small (init) routine relocates the stack to the end of the code segment after the start of the program and releases all remaining memory. Even when this program loads another program, it remains resident. This routine can be useful to applications and can be part of any COM program.

You can assemble this program with either MASM or TASM. You must assemble the source program using an assembler. Let's look at creating the program using the Microsoft assembler. First you assemble the code using MASM:

```
masm testcom;
```

After assembling the program, use the LINK program to create an EXE file:

```
link testcom;
```

When you execute LINK, the following message appears:

```
Warning: no stack segment
```

This message tells you what you already know: The program doesn't contain a stack segment, so it may not function correctly as an EXE file. Simply disregard this message.

If no other warnings or errors were indicated, you must convert the file to a COM file. The EXE2BIN program already mentioned performs this conversion using the following syntax:

```
exe2bin testcom.exe testcom.com
```

Now there may be two files named TESTCOM on the disk. Delete the one named TESTCOM.EXE; you want the TESTCOM.COM file. If all steps were performed correctly, the TESTCOM.COM program can be executed from DOS by simply typing "TESTCOM".

The following command set is required:

```
masm testcom;  
link testcom;  
exe2bin testcom.exe testcom.com
```

Borland's Turbo Assembler (TASM) and the TLINK linker perform the same task as MASM, LINK, and EXE2BIN, but in a slightly different manner. Assemble the TESTCOM.ASM file using TASM as follows:

```
tasm testcom
```

Use the following syntax for linking (this directly links the code to COM form):

```
tlink /t testcom
```

EXE Programs

EXE programs have an advantage over COM programs because they aren't limited to a maximum length of 64K for code, data and stack. However, these files are more complicated. This means that in addition to the program itself, other information must be stored in an EXE file.

Future versions of DOS may enable EXE programs to adapt to innovations, such as multitasking, because it's easier for DOS to estimate memory requirements for EXE programs than for COM programs.

EXE vs. COM

EXE programs contain separate segments for code, data, and stack that can be organized in any sequence. Unlike a COM program, an EXE program loads into memory from disk, undergoes processing by the EXEC function; and then finally begins execution. This is necessary because of the limitations already described for COM programs.

Instead of being limited to loading at a fixed memory location, EXE programs can load at any desired location in memory that's a multiple of 16. Since an EXE program can have several segments, FAR machine language instructions must be used. For example, a main program can be in one segment and call a subroutine in another segment. The segment address must be provided for this FAR instruction in addition to the offset for the routine to be called. The problem is the segment address may be different for every execution of the program. Consulting the CS register doesn't help because only the segment address of the current code segment, instead of the one to which the jump will be made, is stored there.

COM files avoid this problem because the program size is limited to 64K. So, FAR commands are unnecessary. EXE programs solve this problem in a more complex way. The LINK program places a data structure at the beginning of every EXE file that contains (in addition to other things) the addresses of all segments. It contains the addresses of all memory locations in which the segment address of a certain segment is stored during program execution. More specifically, these addresses indicate the addresses of the segment references within instructions. The assembly language code for a FAR jump consists of the following five bytes: the instruction code (one byte), the segment address (one word), and the offset address (one word). The words refer to the location of the jump.

If the EXEC function loads the EXE program, it knows the addresses where the various segments should be loaded. So, it can enter these values into the memory locations at the beginning of the EXE file. Because of this, more time elapses between the initial program call and when the program actually begins execution than for a COM program. The EXE program also occupies more memory than a COM program. The following table shows the structure of the header for an EXE file:

EXE file header		
Address	Contents	Type
00H	EXE program identifier (5A4DH)	1 word
02H	file length MOD 512	1 word
04H	file length DIV 512	1 word
06H	Number of segment addresses for passing	1 word
08H	Header size in paragraphs	1 word
0AH	Minimum number of paragraphs needed	1 word
0CH	Maximum number of paragraphs needed	1 word
0EH	Stack segment displacement	1 word
10H	SP register contents when program starts	1 word
12H	Checksum based on EXE file header	1 word
14H	IP register contents when program starts	1 word
16H	Start of code segment in EXE file	1 word
18H	Relocation table address in EXE file	1 word
1AH	Overlay number	1 word
1CH	Buffer memory	??
??H	Address of passing segment addresses (relocation table)	??
??H	Program code, data and stack segment	??

After the segment references within the EXE program have been resolved to the current addresses, the EXEC function sets the DS and the ES segment register to the beginning of the PSP, which also precedes all EXE programs in memory. Because of this, the EXE program can access the information contained in the PSP, such as the address of the environment block and the parameters contained in the command line (command tail). The stack address and the contents of the stack pointer are stored in the EXE file header and accessed from there. This also applies to the code segment address containing the first instructions of the program and the program counter. After the values have been assigned, the program execution begins.

To ensure compatibility with future DOS versions, an EXE program should terminate by calling interrupt 21H function 4CH.

RAM allocation

Obviously, memory must be available for the EXE program. The EXE loader determines the total program size based on the size of the individual segments of the EXE program. Then it can allocate this amount of memory and some additional memory immediately following the EXE program. The first two fields of the EXE program file header contain the minimum and maximum size of memory required in paragraphs (1-6 bytes).

First, the EXE loader tries to reserve the maximum number of paragraphs. If this isn't possible, the loader tries to reserve the remaining memory, which cannot be smaller than the minimum number of paragraphs. These fields are determined by the compiler or assembler, instead of the linker. The minimum is 0 and the maximum allowed is FFFFH. In most instances, this last number is unrealistic (it adds up to 1 megabyte) but reserves the entire memory for the EXE program.

Now we encounter the same problems as in COM programs. EXE files make poor resident programs, but an EXE program may need to call another program during execution. This is only possible if you first release the additional reserved memory. The following program contains a routine that reduces the reserved memory to a minimum.

You'll find the following program(s) on the companion CD-ROM



TESTEXE.ASM
(Assembler listing)

The program uses separate code, data, and stack segments. You can use this program as a model for other EXE programs. To create an EXE program, assemble it like a normal program with an assembler. Then link it with the LINK program. If the program doesn't contain errors, the LINK program creates an EXE file. The following steps are used to prepare an EXE program from the assembly language source, named TESTEXE.ASM, using the MASM assembler:

```
masm testexe;
link testexe;
```

If you're using the Turbo Assembler TASM, the steps would be as follows:

```
tasm testexe
tlink testexe
```

If all these steps were executed correctly, the program TESTEXE.EXE can be started from the DOS level by typing "TESTEXE".

The Program Segment Prefix (PSP)

We'll end this chapter by discussing the Program Segment Prefix (PSP) which DOS places before every EXE or COM program in memory. The PSP is a remnant of the CP/M era. It contains data DOS needs to manage the program to be executed. The PSP also stores information that's important to programmers, especially parameters supplied by the user when the program is called from the system prompt. While high level language compilers automatically read these parameters at the beginning of the program and write them to predefined global variables, the assembly language programmer must evaluate this information him/herself.

The following table shows the structure and fields of the PSP, many of which remain undocumented (or "reserved") by Microsoft. Most of these fields have been decoded, even though they are useless for practical programming.

Structure of the PSP		
Address	Contents	Type
00H	Interrupt 20H call	2 bytes
02H	Segment address of memory allocated for program	1 word
04H	Reserved	1 byte
05H	Interrupt 21H call	5 bytes
0AH	Copy of interrupt vector 23H	2 words
0EH	Copy of interrupt vector 23H	2 words
12H	Copy of interrupt vector 24H	2 words
16H	Reserved	22 bytes
2CH	Segment address of environment block	1 word
2EH	Reserved	46 bytes
5CH	FCB 1	16 bytes
6CH	FCB 2	16 bytes
80H	Number of characters in command line	1 byte
81H	Command line (CR-LF)	127 bytes

The PSP itself is always 256 bytes long and contains important information for DOS and the program to be executed. Memory location 00H of the PSP contains a DOS function call to terminate a program. This function releases program memory and returns control to the command processor or the calling program. Memory location 05H of the PSP contains a DOS function call to interrupt 21H. Neither of these are used by DOS; they are remnants from the CP/M system.

Memory location 02H of the PSP contains the segment address to the end of the program. Memory location 0AH contains the previous contents of the program termination interrupt vector. Memory location 0EH contains the previous contents of the `Ctrl C` or `Ctrl Break` interrupt vector. Memory location 12H contains the previous contents of the critical error interrupt vector. For each of these memory locations, the program changes one of the corresponding vectors during execution; DOS can use the original vector if it detects an error.

Location 2CH contains the segment address of the environment block. The environment block contains information such as the current search path and the directory in which the COMMAND.COM command processor is located on disk.

Memory locations 5CH through 6CH contain a file control block (FCB). DOS doesn't use the FCB often because it doesn't support hierarchical files (paths) and is also a remnant from CP/M.

The string of parameters that are entered on the command line, following the program name is called the command tail. The command tail is copied to the parameter buffer in the PSP beginning at memory location 81H and its length is stored at memory location 80H. Any redirection parameters are eliminated from the command tail as it's copied to the parameter buffer. The program can examine the parameters in the parameter buffer to direct its execution.

The parameter buffer is also used by DOS as a disk transfer area (DTA) for transmitting data between the disk drive and memory. Most DOS programs do not use the DTA contained in the PSP because it's another remnant from CP/M.



Character Input And Output From DOS

DOS input and output functions can address the keyboard, screen, printer, and serial interface. These functions can be divided into two types: Those carried over from the CP/M operating system and those borrowed from the UNIX operating system. While the two types of functions can be intermingled, to maintain consistency, we recommend using one type of function throughout a program. UNIX type functions use a file handle as an identifier to a device. Because of recent DOS trends to move closer to UNIX, you may want to give the handle functions precedence.

Handle Functions

The handle functions perform file access as well as character input to or output from a device. DOS recognizes the difference by examining the name assigned by the handle. If the handle is a device name, it addresses the device; otherwise it assumes that file access should occur. The table on the right lists the device names. Output and input go to and from the AUX, PRN, and NUL devices. For the device CON, output is sent to the screen and input is read from the keyboard.

Device	Purpose
CON	Keyboard and screen
AUX	Serial Interface
PRN	Printer
NUL	Imaginary device (nothing happens on access)

When DOS passes control to a program, five handles are available for access to individual devices. These handles have values from 0 to 4 and represent the devices listed in the table to the right. The following example shows how to use this table.

Display error message

If a program wants to accept input from the user, this is indicated by the handle function 0 during the call because the standard input device is addressed. Handle 0 normally represents the keyboard, permitting input from the user to the program. Since the user can redirect standard input, you can redirect input to originate from a file instead of the keyboard. This redirection remains hidden from the program.

Handle	Purpose
0	Standard input (CON)
1	Standard output (CON)
2	Standard output for error messages (CON)
3	Standard serial interface (AUX)
4	Standard printer (PRN)

You should first be familiar with some functions used to access any device. Function 40H of interrupt 21H sends data to a device. The function number (40H) is passed in the AH register and the handle is passed in the BX register. For example, to display an error message, the value 2 indicates the handle for displaying the error message (this device cannot be redirected, so handle 2 always addresses the console). The number of characters to be in the error message is passed in the CX register. The characters that constitute the message are stored sequentially in memory, whose segment address is stored in the DS register and offset address in the DX register.

Following the call to the function, the carry flag signals any error. If there's no error, the carry flag is reset. The AX register contains the number of characters that were displayed. If it contains the value 0, then there was no more space available on the storage medium for the message. If the carry flag is set, the error message wasn't sent and an error code is indicated in the AX register. An error code of 5 indicates the device wasn't available. An error code of 6 indicates the handle wasn't opened.

Function 3FH of interrupt 21H reads character data from a device and is very similar to the previous function. Both functions have identical register usage. The function number is passed in the AX register and the handle in the BX register. The number of characters read is passed in the CX register and the memory address of the characters transferred are passed in the DS:DX

register pair. Following the call to the function, the carry flag also signals any error. Again, any error code is passed in the AX register. Error codes 5 and 6 have the same meaning as in function 40H. If the carry flag is reset, then the function executed successfully. The AX register then contains the number of characters read into the buffer. A value of 0 in the AX register indicates the data to be read should have come from a file but this file doesn't contain any more data.

As we mentioned, it's possible to redirect the input or output when accessing DOS. For example, a program that normally expects input from the keyboard can be made to accept the input from a file. So, to avoid having input or output redirected, you can open a new handle to a specific device. This handle ensures the transfer of data to or from the desired device takes place instead of to or from a redirected device.

Use function 3DH of interrupt 21H to open such a device. The function number 3DH is passed in the AH register. The AL register contains 0 to enable reading from the device, 1 to enable writing to the device, and 2 for both reading and writing to the device. The name of the device is placed in memory whose address is passed in the DS:DX register pair. The names must be specified in uppercase characters so DOS can properly identify the device name. The last character of the string must be an end character (ASCII value 0).

Following the function calls, the status is indicated by the carry flag. A reset flag means the device was opened successfully and the handle number is passed back in the AX register. A set flag indicates an error and the AX register contains any error code. The handle is closed using function 3EH of interrupt 21H. The function number is passed in the AH register and the handle number is passed in the BX register. The carry flag again indicates the status of the function call. A set carry flag indicates an error. You can also close the predefined handles 0 through 4 using this function but if you close handle 0 (the standard input device) you can no longer accept input from the keyboard.

Keyboard

The keyboard can perform only read operations. The results of the read operations depend on the mode in which the device was addressed. Here DOS differentiates between raw and cooked. In the cooked mode DOS checks every character sent to or received from a device to determine whether it's a special control character. If DOS finds a special control character, it performs a certain action in response to the character. In raw mode, the individual characters are passed through unchecked and not manipulated. DOS normally operates the device in cooked mode for character input and output. However, you can switch to raw mode within a program (see below).

The best way to illustrate the difference between cooked and raw mode is with an example of reading the keyboard. Suppose that 30 characters are read from the keyboard in cooked mode. As you enter the characters, DOS allows you to edit the input using several control keys. For example, **Ctrl C** and **Ctrl Break** abort the input. The **Ctrl S** keys temporarily halts the program until another key is pressed. The **Ctrl P** keys direct subsequent data from the screen to the printer (until **Ctrl P** is pressed again). The **Backspace** key removes the last character from the DOS buffer. If the **Enter** key is pressed, the first 30 characters (or all characters input up to now if there are less than 30) are copied from the DOS buffer into the input buffer of the program without the control characters.

All characters entered in raw mode (including control characters) are passed to the calling program without requiring the user to press the **Enter** key. After exactly 30 characters, control passes to the calling program, even if you pressed the **Enter** key as the second character of the input.

Screen

To display characters on the screen, handle 1 is usually addressed as the standard output device. Since this device can be redirected, output through this handle can pass to devices other than the screen. However, you cannot redirect the standard error output device (handle 2). So, error messages that pass through this handle always appear on the screen. This handle is recommended only for character display on the screen. The screen is normally addressed in cooked mode; every character displayed on the screen is tested for the **Ctrl C** or the **Ctrl Break** control characters. Since this test slows down the screen output, changing to raw mode occasionally decreases program execution time.

Printer

Unlike the keyboard and screen, printer output cannot be redirected (at least not from the user level). An exception to this rule is redirecting output from a parallel printer to a serial printer. Characters ready to print can be sent to a buffer before they

are sent to the printer. Handle 4 is used to address the standard printer. There are three standard printer devices LPT1, LPT2, and LPT3. Device PRN is synonymous with LPT1. When this handle is opened, the device name is specified as one of the three: LPT1, LPT2 or LPT3.

Serial interface

Much of the information that applies to the printer also applies to the serial interface. For example, serial input and output cannot be redirected to another device (e.g., from a serial printer to a parallel printer). The programmer can use the predefined handle 3 for serial access, through which you can address the standard serial interface (AUX). Handle 3 is used to address the standard serial device. The two are named COM1 and COM2. A PC can have multiple serial interfaces. Only the first two (COM1 and COM2) are supported by DOS. Since the system doesn't know exactly which interface to access during AUX device access, you should open a new handle for access to the specific device.

Errors during read operations in DOS mode are returned to the serial interface in cooked mode. The number returned to the AX register won't match the number of characters actually read. We recommend that you operate the serial interface in raw mode, even if this mode ignores control characters, such as **Ctrl****C** and EOF (end-of-file).

Traditional DOS Functions

The DOS functions for input and output aren't based on the handle oriented functions. If you use these functions you won't need to specify a handle, since each function pertains to a specific device. The various input and output devices and the way in which these functions work with them are listed later.

Keyboard

The seven DOS functions for addressing the keyboard differ in many ways. For example, these functions respond differently to the **Ctrl****Break** key. While some functions echo the characters on the screen, others don't. You can use DOS functions 01H, 06H, 07H, and 08H to read a single keyboard character. The function number is passed in the AH register. Following the call, the character is returned in the AL register.

For DOS function 01H, DOS waits for a keypress if the keyboard buffer is empty. When this occurs, the character is echoed on the screen. If the keyboard buffer isn't empty, a new character is fetched and returned to the calling program. Use DOS function 06H for both character input and output. To input a character, a value of FFH is loaded into the DL register. Instead of waiting for a character to be input, this function immediately returns to the calling program. If the zero flag is set, a character wasn't read. If the zero flag is reset, a character was read and returned in the AL register. The character isn't echoed on the screen.

DOS functions 07H and 08H are used to read the keyboard similar to function 1. Both either fetch a character from the keyboard buffer or wait for a character to be entered at the keyboard. Neither echo the character to the screen. They differ because function 08H responds to **Ctrl****C** but function 07H doesn't. By using function 0BH, a program can determine whether one or more characters are in the keyboard buffer before calling any functions that read characters. After calling this function, the AL register contains 0 if the keyboard buffer is empty, and FFH if the keyboard buffer isn't empty.

DOS function 0CH is used to clear the keyboard buffer. After the buffer is cleared, the function, whose number was passed to function 0CH in the AL register, is automatically called. DOS function 0AH is used to read a string of characters. Again, this function number is passed in the AH register. Also, the memory address of a buffer for the character string is passed in the DS:DX register pair. This buffer is used to hold the character string. The first byte of the buffer indicates the maximum number of characters that may be contained in the buffer.

Function	Task	<Ctrl><C>	Echo
01H	Character input	yes	yes
06H	Direct character input	no	no
07H	Character input	no	no
08H	Character input	yes	no
0AH	Character string input	yes	no
0BH	Read input status	yes	no
0CH	Reset input buffer then input	varies	varies

When this function is called, DOS reads up to the maximum number of characters and stores them in the buffer starting at the third byte. It reads until either the maximum number of characters is entered or the **Enter** key is pressed. The actual number of characters is stored in the second byte of the buffer. Extended key codes, which occupy two bytes each in the buffer, may be entered. The first byte of the pair (ASCII value 0) signifies that an extended key code follows. This means, for example, that for a maximum buffer size of 10 bytes, only five extended characters may be entered.

The table on the right shows how the various functions respond to **Ctrl C** or **Ctrl Break**. It summarizes the individual functions for character input.

Screen output

Three DOS functions are available for character output. DOS function 02H outputs a single character to the screen or standard output device. This character is passed to the DL register. DOS function 06H, which is multipurpose, is also used to output a single character. The character is passed in the DL register. You can see the character, whose value is 255, cannot be output because this indicates the function must perform an input operation. Output using this function is faster than using function 02H because it doesn't test for the **Ctrl C** or **Ctrl Break** keys. DOS function 09H is used for string output. Again, the function number is passed in the AH register. The address of the string is passed in the DS:DX register pair. The last character of the string is a dollar sign. As with function 02H, this function also checks for **Ctrl C** or **Ctrl Break**. Also, the following control codes are recognized:

Code	Character	Operation
07	Bell	Sounds a beep
08	Backspace	Erases preceding character and moves cursor left by one character
10	Linefeed	Moves cursor one line down (LF)
13	Carriage return	Moves cursor to the beginning of the current line (CR)

Printer output functions

DOS function 05H is used to output a single character to the printer. If the printer is busy, this function waits until it's ready before returning control to the calling program. During this time, it will respond to the **Ctrl C** and **Ctrl Break** keys. The function number is passed in the AH register. The character to output is passed in the DL register. The status of the printer isn't returned. Most programmers use the BIOS function instead of the DOS function for printer output because they can specify the exact printer device and determine the printer status using the BIOS version.

Serial interface functions

The two DOS functions for communicating using a serial interface are used for input and for output. Both functions respond to **Ctrl C** and **Ctrl Break**, but they don't return the status of the serial interface or recognize transmission errors. DOS function 03H is used to input data from the serial interface. The character is returned in the AL register. Since the data isn't buffered, it can overrun the interface if the interface receives data faster than this function can handle it. DOS function 04H is used to output data over the serial interface. The character to output is passed in the DL register. If the serial interface isn't ready to accept the data, this function waits until the serial interface is free. Again, most programmers prefer to use the BIOS equivalent functions (see Chapter 3) to perform serial data transmission because of their complete data handling capabilities.

toggling Between Raw And Cooked Modes

We mentioned that it's possible to switch a device from cooked mode to raw mode and then back again. The Pascal and C programs that follow demonstrate how to do this. They use the IOCTL functions which permit access to the DOS device drivers (see Chapter 27 for details on this routine). These routines act as interfaces between the DOS input/output functions and the hardware. The IOCTL functions in these programs tell the CON device driver (responsible for the keyboard and the display) whether it should operate in the cooked mode or in the raw mode.

To demonstrate how differently characters respond in the two modes, the programs switch the CON driver into raw mode first. Then this driver displays a sample string several times. Unlike cooked mode, pressing **Ctrl C** or **Ctrl S** in raw mode has no effect on stopping program execution or text display.

After the program finishes displaying the sample string, the driver switches to the cooked mode. The sample string is displayed again several times. When you press **Ctrl**+**C**, the program stops (Turbo Pascal version). For the C version, you can press **Ctrl**+**C** to stop the program or press **Ctrl**+**S** to pause or continue the display.

Switching between the raw and the cooked mode doesn't occur directly through a function. First the device attribute of the driver is determined. This attribute contains certain information that identifies the driver and describes its method of operation. One bit in this word indicates whether the driver operates in raw or cooked mode. The programs set or reset this bit, depending on the mode you want running the driver.

You'll find the following program(s) on the companion CD-ROM



RAWCOOK.PAS (Pascal listing)
RAWCOOK.C (C listing)

DOS Filters

Filters are programs, routines or utilities that accept input and modify the data for output. Filters also perform these tasks on the operating system level. Characters are passed to these filters as input. Then the filters modify the characters and send them as output. This manipulation takes many forms. Filters can sort data, replace certain data with other data, encode data, or decode data. The table on the right lists the basic filters used by DOS.

FIND	Searches input for a specified set of characters
SORT	Arranges text or data in order
MORE	Formats text display

These filters perform simple redirection of standard input/output. They read characters from the standard input device, manipulate the characters as needed, then display them on the standard output device. Under DOS, the standard input device is the keyboard and the standard output device is the monitor. DOS Versions 2.0 and higher allow the user to redirect the standard input/output to files. So, depending on the standard input device selected, a filter can read characters from the keyboard or from a file. This is possible by using a filter along with one of the DOS handle functions for reading and writing. DOS provides the five handles listed in the table to the right.

0	Standard input	CON (Keyboard)
1	Standard output	CON (Screen)
2	Standard error output	CON (Screen)
3	Standard serial interface	AUX
4	Standard printer	PRN

If the user calls a program from the DOS level, the "<" character redirects input and the ">" character redirects output. In the following example, the input comes from the file IN.TXT instead of the keyboard. The output is written to the file OUT.TXT instead of the screen:

```
sort <in.txt >out.txt
```

SORT

After the user enters the previous command, DOS recognizes that a program named SORT should be called. Then it encounters the expression <IN.TXT, which redirects the standard input. This occurs by assigning the handle 0 (standard input, which formerly pointed to the keyboard) to the file IN.TXT. The expression >OUT.TXT resets handle 1 to the OUT.TXT file instead of the screen. The affected handle is first closed, and then the redirected file is opened. Once the command processor finishes with the command line, it calls the SORT program by using the EXEC function (DOS function 4BH). Since the program called with the EXEC function has all the handles of the calling program available, the SORT program can input/output characters to handles 0 and 1. Where the characters originate isn't important to the program. After the SORT program completes its work, it returns control to the command processor. The command processor resets the redirection and waits for further input from the user.

NOTE

DOS cannot send data from one filter directly to another because it would have to execute both filters simultaneously. However, the current version of DOS doesn't have multiprocessing capabilities. Instead, the following method is used. The input calls the first filter and redirects its output to a pipe file. After the first filter ends its processing, it calls the second filter but redirects its input to the pipe file to read in the output from the first filter. This principle applies to all filters. The pipe file is stored in the current working directory.

Pipes

The filter principle, as supported by DOS, becomes especially powerful through pipes. This is similar to a pipeline used for transporting oil or gas. DOS pipes have a similar function; they carry characters from one program to another and allow various programs to be connected to each other. When this happens, characters output from one program to the standard output device can be read by another program from the standard input device. As in the redirection of the standard input/output, the two programs don't notice the pipelines. The difference between the two procedures is that under redirection of the standard input/output devices, data can be redirected to only one device or file, while the use of pipes allows data transfer to another program.

Combined filters

Pipes allow users to connect multiple filters. The pipe character | is inserted between the programs to be connected. For example, suppose that a text file named DEMO.TXT is sorted and then displayed on the screen in page format. Even though this task appears to be very complicated at first, it can be performed easily using the DOS filters SORT and MORE. SORT sorts the file and MORE displays the file on the screen in page format. Use SORT to tell the command processor to perform these tasks. This filter is told to sort the file DEMO.TXT. The redirection of standard input can be used:

```
SORT <DEMO.TXT
```

After the user enters this command, SORT sorts the file DEMO.TXT and then displays the file on the screen. This display would be much easier to read in page format. Formatted output can be implemented by redirecting the output from SORT to a file (e.g., TEMP.TXT) and displaying this file using the MORE command. The following sequence of commands do this:

```
SORT <DEMO.TXT >TEMP.TXT  
MORE <TEMP.TXT
```

You can use a pipe to connect the SORT filter and the MORE filter, which saves typing time. The following command line sends the output from SORT directly to MORE and immediately displays the sorted file in page format:

```
SORT <DEMO.TXT | MORE
```

Any number of filters can be connected using pipes. DOS always executes these pipelined filters from left to right. It sends the output from the first program as input to the second program, the second program's output as input to the third program, etc. The last program can again force the redirection of the output with the > character so the final result of the whole program or filter chain travels to a file or other device instead of the screen.

Sample programs

"Dump" is a computer term that refers to a way to display the contents of a file in ASCII characters and/or hexadecimal numbers. The following DUMP programs perform this task as a filter. As the contents are displayed in ASCII format, DUMP differentiates between normal ASCII characters (letters, numbers, etc.) and control characters, such as a carriage return, linefeed, etc. These control characters are displayed in mnemonic form (e.g., <CR> for carriage return and <LF> for linefeed). Although this DUMP filter has a fairly simple structure, it can be very useful for quickly examining a file's contents.

The structure of the DUMP program is typical for a filter. Since DUMP displays a maximum of nine ASCII characters and/or hexadecimal codes per line, it asks for nine characters by using the read function from the standard input device. If enough characters aren't available, it reads the available characters. DUMP places these characters in a buffer, then converts the characters into ASCII characters and hex codes. This buffer will accept a complete line of 78 characters. When the buffer processing is complete, the filter uses the handle to write to the standard output device. This process is repeated until no more characters can be read from the standard input device.

We did not include a BASIC version because, as an interpreted language, it isn't suitable for developing a filter that can be called from the DOS level. A BASIC compiler is needed for this task.

You'll find the following program(s) on the companion CD-ROM



DUMPP.PAS (Pascal listing)
DUMPC.C (C listing)
DUMPA.ASM (Assembler listing)



File Management In DOS

The DOS file management functions are among the most basic available to the programmer. However, programmers using high level languages seldom access these DOS functions directly because the languages often have their own methods of file management. This chapter describes how the file functions are organized and how you can use them from higher level languages.

Two Sides Of DOS

The term *file management functions* refers to the functions used to manage files, such as creating, deleting, opening, closing, reading from, and writing to files. Operating systems such as DOS provide the programmer with functions for file management. For example, DOS provides functions that return special file information or rename a file.

One peculiarity of DOS is these functions exist in two forms because of the combined CP/M & UNIX compatibility. For every UNIX compatible file function, there is also a CP/M compatible file function. Versions 2.0 and up of DOS borrowed ideas from this UNIX compatibility.

FCB (File Control Block) functions

The CP/M compatible functions are designated as FCB functions because they are based on a data structure called the FCB (File Control Block). DOS uses this data structure for storing information during file manipulation. The user must reserve space for the FCB within this program. The FCB permits access to the FCB functions which open, close, read from, and write to files. Since the FCB functions were developed for compatibility with CP/M's functions, and since CP/M doesn't have a hierarchical file system, FCB functions don't support paths. As a result, FCB functions can only access files that are in the current directory.

DOS Versions 2.0 and up support handle functions, which were first used in the UNIX environment. However, the UNIX compatible handle functions don't have the problems resulting from FCB functions. As the name suggests, a handle is used to identify the file to be accessed. DOS stores information about each open file in an area that is separate from the program.

Remember the differences between these function groups are related to how these files are created, instead of their actual structures. Files created and edited using FCB functions can cause problems if subsequently accessed by handle functions and vice versa.

The most important fact to remember is to keep the two groups of functions separate when developing high level language programs. In the following sections, we'll take a closer look at each group of functions.

Handle Functions

It's easier for the programmer to access a file using the handle functions than using the FCB functions. With handle functions a programmer doesn't have to use a data structure for file access like the FCB functions do. Similar to the functions of the UNIX operating system, file access is performed using a filename. The filename is passed as an ASCII string when the file is opened or first created. This must be performed before the first write or read operation to the file. In addition to the filename, it may contain a device designator, a pathname, and a file extension. The ASCII string ends with the end character (ASCII code 0). After the file is opened, a numeric value called the handle is returned. Any further operations to this file are performed using this 16-bit handle. For a subsequent read or write operation, the handle, instead of the filename, is passed to the appropriate function.

For each open file, DOS saves certain information pertaining to that file. If the FCB functions are used, DOS saves the information in the FCB table within the program's memory block. When the handle functions are used, the information is stored in an area outside of the program's memory block in a table that is maintained by DOS. The number of open files is therefore limited by the amount of available table space. The amount of table space set aside by DOS is specified by the FILES parameter of the CONFIG.SYS file:

```
FILES = X
```

In DOS Version 3.0, this maximum is 255. If you change the maximum number of files in the CONFIG.SYS file, the change will not become effective until the next time DOS is booted.

```
FILES
```

While the FILES parameter (CONFIG.SYS) specifies the maximum number of open files for the entire operating system, DOS limits the number of open files to 20 per program. Since five handles are assigned to standard devices, such as the keyboard, monitor, and line printer, only 15 handles are available for the program. For example, if a program opens three files, DOS assigns three available File handles and reduces the number of additional available handles by three. If this program calls another program, the three files opened by the original program remain open. If the new program opens additional files, the remaining number of handles available is reduced even further.

Variable access length

Another difference from FCB functions lies in read and write functions. While FCB functions work with records of constant lengths, handle functions specify how many bytes should be read or written. This makes dynamic access to consecutive data records possible.

Besides the standard read and write functions, there is also a file positioning function. This lets you specify an exact location within the file for the next data access. Knowing both a record number and the length of each data record allows you to specify the position to access a particular data record:

```
position = record_number * record_length
```

This function isn't used during sequential file access because DOS sets the file pointer during the opening or creation of a file to the first byte within the file. Each subsequent read or write operation moves the file pointer, by the number of bytes read, towards the end of the file so the next file access starts where the previous one ended.

The following table summarizes the handle functions. For a more detailed description of these functions, see Appendix D, which documents the DOS API functions.

You'll find additional information on the companion CD-ROM



Appendix D on the companion CD-ROM will give you a more detailed description of these handles. Appendix D also documents the DOS API functions.

Function	Operation	Function	Operation
3CH	Create file	57H/01H	Read/Write modifications & date/time of file
3DH	Open file	5AH	Create temporary file
3EH	Close file	5BH	Create new file
42H	Move file pointer/determine file size	5CH/00H	Protect file range against access
43H	Read/Write file attribute	5CH/01H	Release protected file range
56H	Rename file	6CH	Extended OPEN function
57H/00H	Read/Write modifications & date/time of file		

Here are a few general rules to follow when using these functions:

1. Functions that expect a filename or the address of a filename as an argument (e.g., Create File and Open File) expect the segment address of the name in the DS register and the offset address in the DX register. If the function successfully returns a handle, it's returned in the AX register.
2. Functions that expect a handle as an argument expect to find it in the DX register. After the call, the carry flag indicates whether an error occurred during execution. If an error occurs, the carry flag is set and the error code is returned in the AX register.
3. Function 59H of DOS interrupt 21H returns very detailed information about errors that occur during disk operations. This function is available only in DOS Versions 3.0 and higher.

FCB (File Control Block) Functions

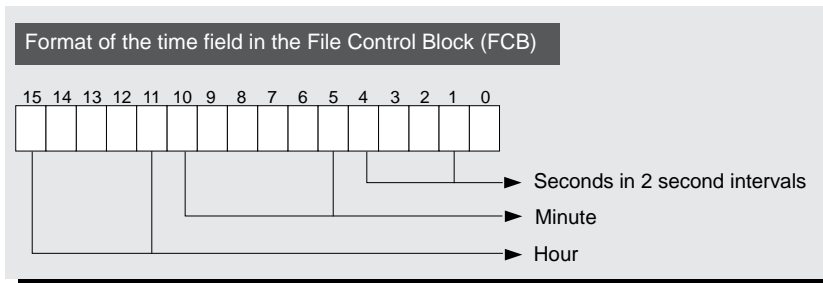
As we discussed, DOS uses an FCB data structure for managing a file. The programmer can use this data structure to obtain information about a file or change information about a file. So, we'll examine the structure of an FCB before discussing the individual FCB functions. The FCB is a 37-byte data structure that can be subdivided into different data fields. The following figure illustrates these fields:

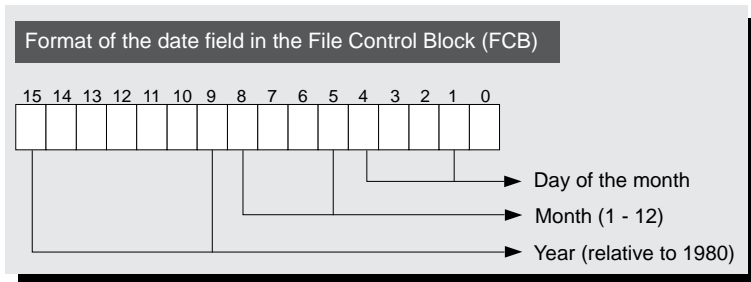
FCB 37-byte data structure					
Address	Contents	Type	Address	Contents	Type
+00H	Device name	1 byte	+14H	Modification date	1 word
+01H	Filename	8 bytes	+16H	Modification time	1 word
+09H	File mode	3 bytes	+18H	Reserved	1 word
+0CH	Current block number	1 word	+20H	Current data record number	1 byte
+0EH	Data record size	1 word	+21H	Data record number for random access	2 words
+10H	File size	2 words			

Notice the name of the file is found beginning at offsets 01H through 0BH of the FCB. The byte at offset 0 is the device indicator, 0 is the current drive, 1 drive A, 2 drive B, etc.

The filename that begins at offset 1 is an ASCII string. It may not contain a pathname since it's limited to 8 characters. For this reason, the FCB functions can access only files in the current directory. Filenames shorter than eight characters are padded with spaces (ASCII code 32). The file extension, if any, occupies the next three bytes of the FCB. At offset 0CH of the FCB is the current number of the block for sequential file access. The two bytes at offset 0EH are the record size. The four bytes at offset 10H are the length of the file.

The date and time of the last modifications to the file are stored beginning at offset 14H of the FCB in encoded form.





An eight-byte data area follows and is reserved for DOS (no user modifications allowed). The use of this area varies from one version of DOS to another.

Following this reserved data area is the current record number which is used with the current block number to simulate CP/M operations.

Random files

The last data field of the FCB is used for a type of access in which the data within the file may be retrieved or written in a non-sequential order. This field is four bytes long. If a record is equal to or larger than 64 bytes, only the first three bytes are used for indicating the current record number. All four bytes of this field are used for records smaller than 64 bytes.

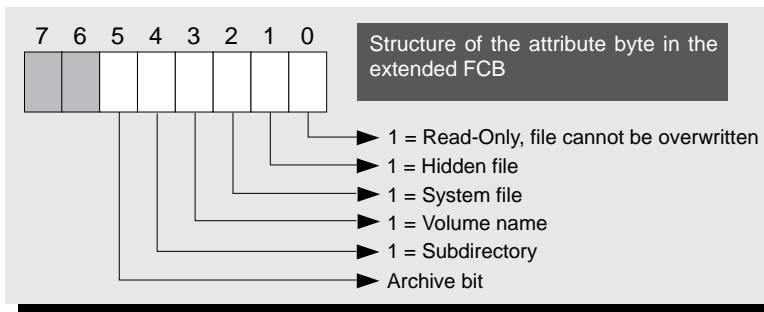
Extended FCB

Besides a standard FCB, DOS also supports the extended FCB. Unlike normal FCBs, extended FCBs access files with special attributes, such as hidden files or system files. They also permit access to volume names and subdirectories (this doesn't mean that you can access files in other directories besides the current directory).

An extended FCB is similar to a standard FCB, but it's seven bytes larger. These seven bytes are located at the beginning of the data structure. So, all subsequent fields are displaced by seven bytes.

Extended FCB data structure					
Address	Contents	Type	Address	Contents	Type
+00H	FF	1 byte	+15H	File record size	1 word
+01H	Reserved(0)	5 bytes	+17H	File size	2 words
+06H	File attribute	1 byte	+1BH	Modification date	1 word
+07H	Device name	1 byte	+1DH	Modification time	1 word
+08H	Filename	8 bytes	+1FH	Reserved	8 bytes
+10H	File extension	3 bytes	+27H	Current data record number	1 byte
+13H	Current block number	1 word	+28H	Data record number	2 words

The first byte of an extended FCB always contains the value 255 and identifies this as an extended FCB. Since this address contains the device number in a normal FCB and therefore cannot contain the value 255, DOS can tell the difference between a normal and an extended FCB. The next five bytes are reserved exclusively for use by DOS. They shouldn't be changed. The seventh byte is a file attribute byte. Refer to the "Floppy Diskette And Hard Drive Structure" section of Chapter 14 for the details of the file attribute byte.



Now that you're familiar with the FCB structures, the next section focuses on using FCBs for accessing files.

FCB and file access

Before accessing a file, an FCB must be built in the program's memory area. The area can be reserved within the data segment of the program or by allocating additional memory using another DOS function (see Appendix D).

Although it's possible to write the data directly into the FCB, it is better to use one of the appropriate DOS functions to do this. For example, to set the filename in the FCB you can use DOS function 29H. The function number is passed in the AH register. The address of the FCB is passed in the ES:DI register pair. The address of the filename is passed in the DS:SI register pair. The filename is an ASCII string terminated by the end character (ASCII code 0). The AL register contains flags for converting the filename and are discussed in more detail in Appendix C.

Open FCB

After the FCB is properly formatted, the file can be opened or created using a DOS function. When this happens, DOS stores information about that file, such as the file size, date and time of file creation, etc., in the FCB. At this point the FCB is considered opened.

By default, the record length is set to 128 bytes when the FCB is opened. To override this record length, store the desired record length at offset 0EH of the FCB after it's opened. Otherwise, the default length will be used.

DTA

For record lengths greater than 128 bytes, the record buffer, also known as the DTA (Disk Transfer Area), must be moved to accommodate the longer record size. Usually DOS builds the DTA in the PSP (Program Segment Prefix). Accessing the file using the default DTA for a record length greater than 128 bytes would overwrite some of the other fields in the PSP.

The most convenient way to select a new DTA is to reserve the space in the program's data segment. To change the address of the DTA, use DOS function 1AH. The address of the new DTA is passed in the DS:DX register pair. Since DOS assumes that you've set aside an area large enough to accommodate your largest record length, you don't have to specify the new length.

File access

For sequential file access, processing begins at the first record in the file. DOS maintains a record pointer in the FCB to keep track of the current record within the file. Each time the file is accessed, DOS advances the pointer so the second, third, fourth, etc. record is processed sequentially.

For random file access, the records can be processed in any order. The position of each record relative to the beginning of the file determines its record number. This record number is then passed to DOS to access a specific record. The last field of the FCB is used to specify the record number to DOS.

It's also possible to change from sequential access mode to random access mode and vice versa, since processing depends on a specific DOS function to access the file. There are actually two sets of independent functions, one for sequential access and one for random functions.

The following table lists all the FCB functions of DOS interrupt 21H. A more detailed description of the functions is found in Appendix D which you'll find on the companion CD-ROM.

FCB functions of DOS interrupt 21H			
Function	Task	Function	Task
0FH	Open file	21H	Random Read (of record)
10H	Close file	22H	Random Write (of record)
13H	Delete file	23H	Determine file size
14H	Sequential read	24H	Set record number for random access
15H	Sequential write	27H	Random read (one or more records)
16H	Create file	28H	Random write (one or more records)
17H	Rename file	29H	Enter filename into FCB
1AH	Set DTA address		

The following are some basic rules about these functions:

Using the FCB functions, you can access several files, each with their own unique FCB. To tell DOS which file should be accessed, pass the address of the file's FCB in the DS:DX register pair.

Most of the functions return an error code in the AL register or the value zero if the function was successfully completed. For functions that open, close, create, or delete a file, a code of 255 is returned if an error occurs. The other functions return specific error codes. More detailed information about these errors can be determined by calling DOS function 59H but this is available only in DOS Versions 3.0 or later.

You'll find additional information on the companion CD-ROM



Appendix D on the companion CD-ROM provides a more detailed description of these functions.

Handles versus FCBs

Now we'll briefly discuss the advantages and disadvantages of the individual functions. If you want to convert a program from the CP/M or UNIX operating systems into DOS, the choice will be easy. However, if you want to develop a new program under DOS, the following explanations should help you determine which set of functions to use.

Handles

There are two main advantages to using handle functions. The first is the ability to access a file in any subdirectory of the disk. The second is the handle functions aren't limited to the number of FCBs that can be stored in a program's memory space.

There are also several additional considerations. You can access the name of a disk drive only by using an FCB. When the FCB is opened, you can easily determine its file size and the date of the last modification. The handle functions automatically provide an area large enough to accommodate the records in the file.

As you can see, there are arguments for and against using either the FCB functions or the handle functions. For future versions of DOS, the handle functions will become more important and the importance of the FCB functions will diminish. This is reason enough to use the handle functions for your new program development.

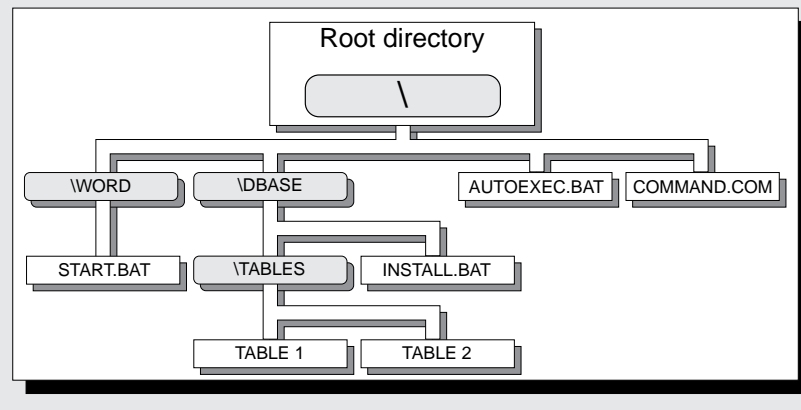
22

Accessing DOS Directories

Two groups of DOS functions are used to work with directories. The first group is used to manipulate the subdirectories and the second group is used to search for files on the mass storage devices.

DOS Version 2.0 introduced subdirectories. A mass storage device could be logically divided into smaller subdirectories, which could also be divided. This organization creates a directory tree.

Example of a directory tree



In this directory tree, the names and numbers of subdirectories are dynamic. There must be a way to add, change, and delete entries on the tree. Other functions must be available to set the current directory so a complete pathname isn't required for all file accesses. At the user level, the MD, RD and CD commands can be used to make a directory, remove a directory and change a current directory. These commands are performed internally with functions 39H, 3A and 3BH of DOS interrupt 21H.

All three functions use identical calling conventions. The function number is passed in the AH register. The address of the path is passed in the DS:DX register pair. The path is a string and may be a complete path designation, including a preceding drive letter followed by a colon (a device name) and terminated by ASCII code 0. If the device name is omitted, the current device is the default.

Following execution, the carry flag indicates the return code. If the carry flag is reset (0), then execution was successful. If the carry flag is set, then an error occurred and the error code is passed back in the AX register.

Function 39H creates or makes a new directory (Make Directory function). The name for the new directory is specified as the last element in the path. An error will be returned by the functions if one or more of the directories specified in the path doesn't exist, if the new directory name already exists, or if the maximum number of files in the root directory is exceeded.

Function 3AH deletes or removes a directory (Remove Directory function). An error will be returned by the function if the target directory isn't empty or the specified directory doesn't exist in the current path.

Function 3BH changes the current directory (Change Directory function). An error is returned if the directories named in the path don't actually exist.

Function 0EH sets the default disk drive. Besides the function number in the AH register, only the device code of the new current device must be passed in the DL register. Code 0 represents the device A, code 1 represents device B, code 2 represents device C, etc.

Directory specification

Before specifying the current directory using function 3BH, sometimes you must find the current directory. DOS provides function 47H for this purpose. Since it can return the path of the current directory for any device, the device number must be passed to the function. If this is the current device, the value 0 must be passed in the DL register. For all other devices, the value 1 must be passed for drive A, 2 for B, 3 for C, etc.

Besides the device code, the function must also have the address of a 64-byte buffer within the user program. The DS register contains the segment and the SI register contains the offset address of this buffer. After the function call, this buffer contains the path designation of the current directory, terminated with the end character (ASCII code 0). The path designation cannot be preceded by the device name or the \ character. If the current directory is the root directory, the buffer contains only the end character. If a device code unknown to DOS was passed during the function call, the carry flag is set and the AX register contains the error code 0FH.

Let's consider the functions for searching for one or more files in the current directory on the current device. Again you can see the connection between handle and FCB functions. Two function groups are used to search for files. The group of FCB functions limit the search to files in the current directory of a certain device, while handle functions allow you to search for files in any directories of any devices. The term "handle" functions isn't really appropriate for these functions because they aren't addressed with a handle. This designation originated with the introduction of subdirectories (and therefore the handle functions) in DOS Version 2.0. Version 1.0 offered only the FCB functions.

How the search function works

Although the handle and FCB functions have different capabilities, they are very similar. Both of them access functions called FindFirst and FindNext. FindFirst is called only once for each file because it initiates the search. This function expects the caller to pass the file's name and attribute (more on this later).

The search name can be conveyed as a filename with a path (e.g., C:\DOS\LETTER.DOC) or without a path (e.g., LETTER.TXT). Wildcard characters (* and ?) can be used to search for patterns instead of specific files. Similar to the DIR command used in DOS, FindFirst and FindNext can display all the files in a directory using these wildcards.

Function	Assignment
11H	FindFirst (FCB)
12H	FindNext (FCB)
4EH	FindFirst (Handle)
4FH	FindNext (Handle)

Regardless of the search name, FindFirst returns only the first filename found (if it exists in the indicated directory). This filename corresponds to the name or name pattern. All additional files can be found through subsequent calls of FindNext; with each call of this function, the next filename that fits the search pattern is returned.

The file attribute's role

The functions listed interact with the standard file attributes as assigned under DOS (see the table below for more information). The bits in the attribute byte specify different attributes. For example, when bit 4 is set, the directory entry is viewed as a subdirectory rather than a file. When bit 2 is set, this indicates that the file is hidden, and won't be visible when you call the DIR command from DOS.

After finding an entry, the attribute byte can be read for set attributes. Both FindFirst functions expect to receive a search attribute, which identifies the files to be found. These attributes don't include the read-only attribute or the archive bit. If the search attribute contains 0, all normal files are displayed whether their read-only and archive bits are set.

Directory entries that describe hidden files, system files, volume names, and subdirectories are treated differently. These are excluded from the file search if the corresponding bit in the search attribute isn't set. For example, if you set bit 4 in the search attribute, all subdirectories are returned as the result.

Attribute byte in file search					
Bit	Attribute	Meaning if set	Bit	Attribute	Meaning if set
00H	Read-only	File is read-only	04H	Subdirectory	Entry is a subdirectory and not a file
01H	Hidden	File is hidden (invisible to DIR)	05H	Archive	Entry is a subdirectory and not a file
02H	System	File is part of operating system	06H	Reserved	Reserved for later implementation
03H	Volume label	Entry is volume label and not a file	07H	Reserved	Reserved for later implementation

This method cannot exclude all normal files from the search in order to search exclusively for hidden files or subdirectories. However, you can evaluate the file attributes of the files that were found and only use the files that have the desired attribute flag set.

Searching For Files Using FCB Functions

This method of searching for files uses functions 11H and 12H. By using these functions, you can search for files with a fixed name or an extension. Function 11H finds the first file in the current directory. Function 12H finds any additional files. The FCBs are important because they mediate between the calling program and the two functions. Let's see how we can search for files in a directory:

First, the program must reserve space for two FCBs. This is done either by reserving memory in the data area of the program or by requesting memory from DOS using function 48H. The programmer can use either normal or extended FCBs, which are capable of searching for files with special attributes (system or hidden), volume names and subdirectories. The filename for which the search will be made is specified in one of the FCBs. DOS places the name of the file(s) that it finds in the other FCB. The two FCBs are identified by their names Search FCB and Found FCB.

The address of the Found FCB must be passed to DOS using function 1AH. When this function call occurs, the Found FCB becomes the new data transmission area (DTA). This area is important for these two functions as well as all other functions that transfer data between computer and disks. Therefore, function 2FH should determine the address of the current DTA before activating the new DTA. When the file search ends, the DTA can be restored to its original status using function 1AH.

After the DTA is set to the Found FCB, place the name of the file you're looking for into the Search FCB. For a more general search, you can use the wildcards * and ?. You can transfer the filename directly or transfer it using function 29H. If you want to search through all files, use the filename *.*. If an extended FCB is used, you may insert an additional value into the attribute field of the Search FCB to limit the search to files with only certain attributes (see the preceding pages of this chapter for more information on the various attributes).

This concludes the preliminary work. The file search can begin with the current directory. For this purpose, function 11H is called with the function number in the AH register, the segment address of the Search FCB in DS, and the offset address in the DX register. If the system finds a file with the indicated name, the AL register contains the value 0 after the function call. If the filename wasn't found, the AL register contains a value of 255.

The found filename and its attributes (if extended FCBs are used) can be read from the Found FCB. For additional searches, function 12H (not function 11H) is called. Function 12H's register contents during call and return are similar to function 11H. If it returns the value 255 in the AL register during one of the calls, the search has ended.

Sample programs

The FF.ASM program demonstrates FCB file searching by using FCB functions 11H and 12H. The assembled program is a COM program instead of an EXE program. FF represents FileFind because the program searches for a certain file, or a group of files, in all the directories of a drive. It shows that the FCB functions can be used for a file search in various directories by specifying the search directory as the current directory before the search begins.

You'll find the following program(s) on the companion CD-ROM



FF.ASM (Assembler listing)

Call FF by using the following syntax:

```
ff [dr:]filename [+|-|=date]
```

Let's look at these parameters:

Parameter	Purpose
[dr:]	This represents the optional drive specifier. FF defaults to the current drive.
filename	This represents the filename or pattern you want to find. This parameter can include wildcards. If you type FF.ASM as the filename, FF will search for only the file FF.ASM. If you type *.ASM, FF searches for all files with ASM extensions, in all directories of the current drive.
+date	Lists files created or last modified after the specified date.
[-date]	Lists files created or last modified before the specified date.
[=date]	Lists files created or last modified on the specified date.

The following command finds all TXT files created after January 10, 1995:

```
ff *.txt +1-10-1995
```

Searching For Files Using Handle Functions

Working with handle functions is easier than working with the FCB functions. Two of these functions are used to search for the first file (the 4EH function) and subsequent files (the 4FH function). Both functions return the information to the DTA. Therefore, the DTA should be moved into an area accessible to the current program before calling either of these functions. This area must have at least 43 bytes available. As mentioned with the FCB functions, the DTA should be restored to its original address after the search ends.

During the call of the 4EH function, the function number is passed in the AH register, the attribute in the CX register, and the address of the file to be found in the DS:DX register pair. The filename is a series of ASCII characters followed by an end character (ASCII code 0). In addition to a device name, you may also add a complete path designation and the wildcard characters * and ?.

If a path isn't specified, DOS assumes that the search should be performed in the current directory of the indicated device. If a device isn't specified, the search continues on the current device. After the function call, the carry flag indicates whether a file was found. If the file couldn't be found, the carry flag is set and the AX register contains an error code. An error code of 2H is returned if the indicated path does not exist. If a file couldn't be found, an error code of 12H is returned. If the carry flag is reset, the DTA contains the information about the file that was found. The following table shows its structure:

Function 4FH performs any subsequent searches. The function number is passed in the AH register; additional parameters aren't necessary. The carry flag indicates whether the current directory contains any additional files that may apply to the search.

Sample programs

Two Pascal programs, two C language programs, and one BASIC program are listed. These dual implementations demonstrate how you can indirectly call DOS functions 4EH and 4FH. Many languages, such as Turbo Pascal, Microsoft C, and Turbo C, include FindFirst and FindNext functions in their libraries. These functions eliminate the need for direct calls to functions 4EH and 4FH and also provide some other help for the programmer. (We'll discuss this in more detail later.)

DTA structure		
Address	Contents	Type
00H	Reserved for DOS	21 bytes
15H	Attribute of file found	1 byte
16H	Time of last modification	1 word
18H	Date of last modification	1 word
1AH	Low word of file size	1 word
1EH	High word of file size	12 bytes

QuickBASIC contains only a rudimentary FILES command but not a library. The FILES command can display only the current directory's contents, without making this information available to the program. Direct calls to functions 4EH and 4FH are unavoidable in BASIC.

Program logic

All five programs are constructed according to the same basic framework. First the main program reads the command to determine which filenames should be displayed. If the program name was entered without parameters, the program defaults to the "*".*" wildcards. If the user entered a parameter after the program name, the program uses that as the pattern for file display. Entering more than one parameter ends the program and displays an error message.

The program passes the name of the file to be displayed and the attribute of the files to be displayed to the DIR function (or procedure). All the attributes are set to include volume name, subdirectories, system files, and hidden files. Although the attribute byte cannot be changed by the user, this byte setting can be altered from within the program code. In the three programs, which directly communicate with functions 4EH and 4FH, the SetDTA procedure/function sets the DTA to a local data structure. SetDTA calls DOS function 1AH to move the DTA to the indicated address.

The data structure intended for the DTA exists in all three programs as type DirStruct. This structure represents a record in which the various fields are found by functions 4EH and 4FH. After DTA's initialization, the DIR procedure/function calls the ScreenDesign procedure/function. This routine creates a window on the screen, in which the directory will be displayed.

FindFirst begins the search, after which a loop continuously calls FindNext until no other files are found. The PrintData procedure/function displays these files on the screen. This procedure/function expects DirStruct as an argument, specifying where DOS stored information about the file. This information is decoded and displayed on the screen. The program and program listing for the DIRB.BAS BASIC version of this directory lister program is on the companion CD-ROM.

You'll find the following program(s) on the companion CD-ROM



DIRB.BAS (BASIC listing)

Direct calls to functions 4EH and 4FH in Pascal and C

The DIRP1.PAS and DIRC1.C Pascal and C programs perform direct calls to functions 4EH and 4FH. Predefined FINDFIRST and FINDNEXT procedures/functions aren't used.

Turbo Pascal includes a statement that easily defines any part of the screen as a window. The C language must use functions of BIOS interrupt 10H to scroll the directory window up, one line at a time. The C program contains a function called PRINT. This function is similar to PRINTF() except that PRINT accepts the string's position on the screen and the color of the output. This display information can then be written directly to video RAM.

You'll find the following program(s) on the companion CD-ROM



DIRP1.PAS (Pascal listing)
DIRC1.C (C listing)

Using the predefined functions/procedures in Pascal and C

The DIRP2.PAS and DIRC2.C programs use the predefined FindFirst and FindNext functions from Turbo Pascal or the various C compilers supported by our example programs. This simplifies directory display because DTA positioning is bypassed.

Both programs use the 4EH and 4FH DOS functions, as well as a special data structure for storing file information. Turbo Pascal's SearchRec structure is defined in the DOS unit, along with FindFirst and FindNext. The DOS unit also defines various constants, which can be set with flags for controlling file attributes.

You'll find the following program(s) on the companion CD-ROM



DIRP2.PAS (Pascal listing)

Unfortunately, the SearchRec structure combines the time and date fields into a long integer, which requires a little trick to divide the individual data elements. The DIRP2.PAS program demonstrates how this is done. Also, notice that FindFirst and FindNext in the Pascal version are procedures rather than functions. So, the program must check the DosError global variable after the function call to ensure that the file was found. After calling functions 4EH and 4FH, Turbo Pascal stores the contents of the AX register in DosError. If this variable contains 0, the file was found.

The C program DIRC2.C encounters some problems because of differences between the Microsoft and Borland compiler libraries—especially those relating to DOS API interfaces. API is not yet enforced by ANSI standards, so compiler manufacturers can set their own standards for API. Thus, FindFirst and FindNext retain different names in the two different C compiler families, and operate using different data structures with different field names.

The DIRC2.C program can be compiled and linked by both compilers. Macros include the device-dependent commands needed by the respective compilers.

You'll find the following program(s) on the companion CD-ROM



DIRC2.C (C listing)



Date And Time

The AT realtime clock provides the date and time for BIOS routines and files. Four DOS functions, which have been available since the release of Version 1.0 of DOS, can be used to access time and date.

DOS and the AT realtime clock

In DOS Versions 3.2 and lower, these four functions passed information to and from DOS environmental variables. However, in DOS Versions 3.3 and higher, these functions passed the same information to and from the battery operated realtime clock. The time and date information is still available, even if the user switches off the computer. The time and date are passed using the BIOS functions described in Chapter 15.

This is also possible in older versions of DOS. However, the system must contain a built-in clock driver because DOS will communicate only with an onboard device driver named \$CLOCK device driver.

Getting and setting the date

Functions 2AH and 2BH control the current date.

Function 2AH (get date): *Get date*

Placing function number 2AH in the AH register returns the current date information in the processor registers listed in the table to the right.

Interrupt 21H, function 2AH

Function 2BH (set date): *Set date*

Function 2BH places date information in the same registers used by function 2AH. This is useful for changing a creation or modification date in a file.

The table on the right shows that function 2BH requires the actual date (but not the day of the week). This is the only information DOS needs.

Remember that when working with this function, the earliest date allowed by the system is January 1, 1980 (01-01-1980). If an error occurs, check the AL register. If this register contains a value of 0, the data supplied in the other registers is valid. If the AL register contains a value of 255, the data couldn't be read.

Getting and setting the time

Functions 2CH and 2DH are similar to functions 2AH and 2BH, except that 2CH and 2DH handle time access.

Function 2CH (get time): *Get time*

Function 2CH reads the current time in the form shown in the following table. Place function number 2CH of DOS interrupt 21H in the AH register to read the following registers:

DOS date and time functions	
Function	Task
2AH	Read date
2BH	Set date
2CH	Read time
2DH	Set time

Output registers: Function 2AH	
Register	Contents
AL	Day of week*
CX	Year
DH	Month
DL	Day
*0=Sunday, 1=Monday, etc.	

Input registers: Function 2BH	
Register	Contents
AH	2BH
CX	Year
DH	Month
DL	Day

Output registers: Function 2CH	
Register	Contents
CH	Hour
CL	Minute
DH	Second
DL	Hundredths of a second

Function 2DH: Set time

Function 2DH sets the time. The AL register indicates whether the time is valid (e.g., a 36:48 setting generates a value of 255, which is an error).

Output registers: Function 2DH	
Register	Contents
AH	2DH
CH	Hour
CL	Minute
DH	Second
DL	Hundredths of a second

BIOS or DOS?

You can use either BIOS or DOS to access time and date. Both methods have their advantages, and both methods are available on any given PC system. Which functions you access depends on your preferences.

24

RAM Management

One of the basic tasks an operating system must perform is managing the RAM (Random Access Memory). This is where all the various system components (device drivers, TSR programs, applications, etc.) come together. Since each of these components needs some of the memory, DOS must ensure they work together. This prevents the components from overwriting each other and a single component from manipulating all the memory.

In this chapter we'll discuss the DOS memory management functions and how they work together to keep your RAM organized.

DOS RAM Management

The RAM management capabilities of DOS are based on the principle of using a DOS function to allocate a memory block of a predetermined size. This memory block remains allocated to the program that requested it until it's freed by another DOS function. Then the block can be used by other programs.

The table on the right shows the four different functions for RAM management in DOS. The functions for allocating and freeing RAM are used by application programs, but they are also used by DOS itself in the form of EXEC loaders. When a program must be loaded and started, the EXEC loader reserves the RAM block, in which it will later load the program.

DOS functions for RAM management	
Function	Purpose
48	Allocate memory
49H	Free memory
4AH	Change size of a memory block
58H	Read/set memory management model

The amount of memory allocated depends on the type of program to be loaded. COM programs will reserve the entire RAM. The amount of memory required for an EXEC program is taken from the header of the EXE file. The EXEC loader can load the file only if a large enough block of RAM is available. If enough RAM cannot be found, the EXEC loader stops and displays the following error message:

```
Insufficient memory
```

Besides the EXEC loader, the application program itself can request memory from DOS. During execution, many programs will need more memory than they were given when they were initially loaded. This happens because they aren't able to determine the values of all variables or the sizes of all buffers at the time they are loaded. This information is usually determined by user input while the program is running.

Memory allocation with function 48H

If you're programming in a high level language, you don't have to worry about dynamic memory allocation. This is handled by the heap. However, assembler programmers must go to DOS when they need to allocate memory. This is done with function 48H. This function call requires the function number in the AH register and the size of the desired memory block in the BX register.

The size of the block is given in the number of paragraphs instead of the number of bytes. Since a paragraph consists of 16 bytes, allocated memory blocks will always be in multiples of 16 bytes. The smallest possible memory block that can be allocated is 16 bytes (BX = 1) and the largest is 1 Meg (BX = FFFFh).

However, requesting an entire megabyte from DOS isn't realistic because the processor doesn't even have that much RAM available in real mode. Actually, you cannot obtain 640K, 512K or even 400K from DOS because a large portion of RAM is used by the DOS programs and the DOS device drivers. TSR programs, which remain resident in memory (i.e., they don't free the memory allocated to them when they are started) also occupy a lot of memory.

Since DOS cannot always fulfill the requests it receives with function 48H, you must check the carry flag after the function call. If this flag is set, then the requested amount of RAM wasn't available. The BX register will then contain a value that corresponds to the actual amount of remaining RAM. This value is also given in paragraphs, so you must multiply it by 16 to obtain the number of bytes.

If the program can be run with this amount of memory, then function 48H is called again with this value in the BX register. All of the RAM that's still available will then be allocated to the program.

After this function call, the program will discover the carry flag isn't set. This means the AX register will contain the segment address of the memory block that DOS has reserved. This block is now completely under the control of the application for which it was allocated.

The application must always remember the size of the block. Depending on the amount of memory allocated, you may or may not be able to use the entire segment at the address returned by function 48H. For example, if you asked for only one paragraph, you'll have a memory block such as AX:0000 through AX:000F. This corresponds to only the first 16 bytes of the segment. If you asked for more paragraphs, the memory block will be correspondingly larger. It will always start at the segment address returned by the function and at offset address 0000H.

What happens if you request 4096 paragraphs? This corresponds to 64K, which would be the entire memory segment from AX:0000 through AX:FFFF. If you ask for more than 4096 paragraphs, the memory block will extend into the next segment.

In all cases, the application that now owns the memory block can address only the portion of the segment that belongs to it. Anything outside of this block may belong to another program or be allocated to another in the future. If two applications try to use the same part of the RAM, the system will usually crash.

Determining the amount of available RAM

Because of the way it works, function 48H can also be used to check the current amount of free RAM. This is the only DOS function that performs this task.

To force DOS to return the amount of free memory, we'll make a request that it can't possibly fulfill. We'll pass the value 0FFFFH in the BX register, thereby requesting a 1 megabyte memory block. Although it's impossible for DOS to allocate a block of this size, the function result in the BX register indicates the number of free paragraphs. From this value, we can easily calculate the number of free bytes remaining.

Releasing memory with function 49H

Once an allocated memory block is no longer needed, it must be released with function 49H. Before a program ends, all the memory allocated with function 48H must be released. Otherwise, the operating system will think that this memory is allocated to an application and other programs won't be able to use it. This can significantly reduce the amount of available memory and you may not be able to load the other programs you want to run. Only rebooting the computer can free this memory.

This function call requires the function number in the AH register and the segment address of the memory block to be freed in the ES register. This corresponds to the value that was returned to the AX register when the block was allocated with function 48H.

If the memory block was successfully released, function 49H returns with a cleared carry flag. DOS again has control of the memory block, which can be allocated to another application. If the function returns with the carry flag set, then the memory block couldn't be released because of an error. Various errors can cause this condition. The error code that describes what occurred is returned to the AX register.

An error code of 9 indicates that an incorrect segment address was given. This means that at the given address, a memory block allocated by this program wasn't found. It's possible the program did own a memory block at this address, but that it has already been freed with a previous call to function 49H. In this instance, you should determine whether you've inadvertently tried to free the same block twice.

Error code 7 indicates that a problem occurred within DOS's internal memory manager. The cause of this problem is usually a program that has tried to operate outside of its reserved memory blocks. If you encounter this error code, you should end your program and recommend a soft reboot to the user.

Changing the size of memory blocks

The third memory management function is function 4AH. This function changes the size of a previously allocated memory block. You can choose to make the block larger or smaller. However, you may not be able to increase the size of the block if there isn't enough memory.

To call function 4AH, pass the function number to the AH register and the segment address, of the memory block to change, to the ES register. The new size of the memory block is passed to the BX register. This value must be given in paragraphs. The contents of the registers after the function call will be the same as with function 48H. If the function fails due to lack of memory, it returns the maximum block size in the BX register. This is only important when increasing the size of a block. You should always be able to make a block smaller.

Where Does Memory Come From?

In DOS Versions up to 5.0, the memory that DOS allocates via function 48H always originates from the Transient Program Area (TPA). This refers to a region of memory that extends from the end of the resident DOS kernel at the beginning of the RAM all the way to the 640K limit. If a system has less than 640K, the TPA ends there. However, these systems are very rare today.

The RAM usually ends at the 640K limit. Beyond this limit is the memory reserved for video RAM, the ROM-BIOS and ROM expansions. The page frame of an EMS card is also stored in this memory region. Systems equipped with 1 Meg or more are usually configured with 640K of RAM below the 640K limit and the rest of the memory starts at the 1 Meg limit.

More memory with Upper Memory Blocks (UMBs)

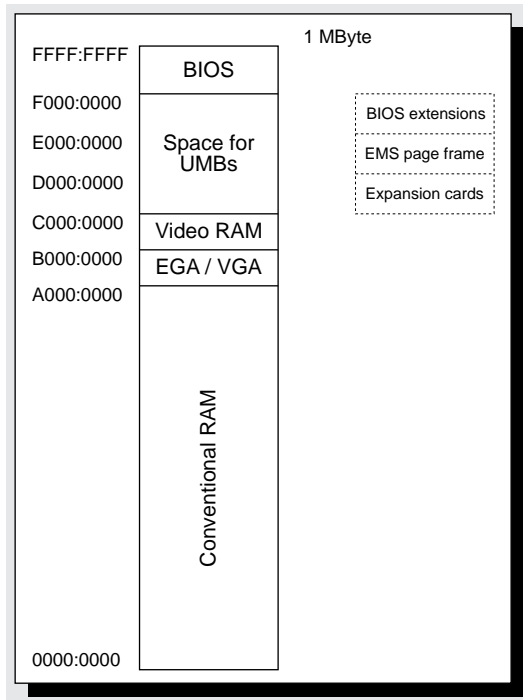
Memory above the 1 Meg limit and memory between 640K and 1 Meg, cannot be directly addressed by DOS in real mode unless you're using DOS Version 5.0. This DOS version supports using Upper Memory Blocks (UMBs), which can be located anywhere above the 640K limit. The region from 640K to 1 Meg usually isn't completely occupied by the video RAM, the ROM-BIOS or other expansions. So, DOS 5.0 can use and allocate the unused regions as normal blocks of RAM.

Memory in the region between 640K and 1 Meg is extremely valuable to DOS programs, because it can be addressed in real mode. With DOS 5.0, a program cannot tell the difference between memory above and below 640K. As long as the memory doesn't extend beyond the 1 Meg limit, it's considered the same. Add UMB memory to your system with:

- The proper software configuration, such as a NEAT chip set, which creates RAM in the UMB region out of extended memory.
- A special UMB card that's equipped with its own RAM, which is then added to the UMB region of the system's addressable memory.
- Memory management programs that move extended memory to the UMB region, such as EMM386.EXE, 386Max, or QEMM.

The last option is becoming more popular since DOS 5.0 was introduced because this version includes the device driver for EMM386.EXE.

Upper memory blocks (UMBs) in the addressable memory of the PC



The amount of memory available for UMBs depends on how much memory is occupied by video RAM, expansion cards and the ROM-BIOS. Many systems will have up to 260K available, while others may have less than 64K. Unlike RAM below 640K, this memory is often fragmented. So UMBs must make room wherever they can in the "holes" between 640K and 1 Meg.

Before DOS programs can use UMBs, either the command:

```
DOS=HIGH,UMB or: DOS=LOW,UMB
```

must be executed in the CONFIG.SYS file. After finding one of these commands, DOS looks for an XMS driver. DOS doesn't actually prepare the memory where the UMBs will be located. Instead, a device driver, such as EMM386.EXE (see above), performs this task. This is useful because different UMB cards have different ways of setting up the UMB memory. So, DOS can leave the setup to the driver that relates to the specific hardware. These drivers must adhere to the XMS standard. This standard defines how DOS allocates UMBs. So, DOS has control over all RAM allocation from the time the system is started.

Special DOS commands, such as DEVICEHIGH and LOADHIGH, are used to load device drivers or DOS programs into a UMB, as long as an appropriate one can be found. Since the UMB memory can be very fragmented, this may be difficult to do with larger programs. Enough memory may be available, but it's split into two or three separate regions in between the video RAM and the ROM-BIOS or something else. Because of this, small TSR programs and device drivers can easily be placed in UMB memory.

UMBs in routine memory management

UMB memory isn't used only when allocated by DOS programs or device drivers with DEVICEHIGH and LOADHIGH. UMBs are also included in routine memory management. This means that a call to function 48H can return a UMB. You can recognize a UMB by a segment address greater than A000H, which corresponds to the 640K limit.

Four subfunctions of DOS function 58H enable you to influence the way UMBs are included in routine memory management. The first two subfunctions, 00H and 01H, have been available since DOS Version 2.0. Subfunctions 02H and 03H were introduced with Version 5.0.

Subfunction 03H allows you to determine whether function 48H should consider using UMBs when allocating memory. This is useful for programs that must allocate a lot of additional memory after they're called.

Subfunction 03 is called with the function number 5803H in the AX register and either 0 or 1 in the BX register. A value of 1 indicates that UMBs will be included in memory allocation; 0 indicates that all memory will be allocated below the 640K limit.

Before making this function call, you should check the status of the UMB memory so you can restore it to its original condition after your program ends. Subfunction 02 will do this for you. This subfunction call only needs the function number 5802H in the AX register. After this function call, the AL register will contain a value of either 0 or 1. These values have the same meaning they have with subfunction 03H.

Remember these two subfunctions are only supported under DOS Versions 5.0 and higher and the DOS UMB command must be run in the CONFIG.SYS file for them to work properly. After calling either of these subfunctions, you should check the contents of the carry flag. If the carry flag is set, then one of the two conditions already described hasn't been met and UMBs cannot be supported. If you want to use UMBs in your program, try working with small memory blocks. You probably won't find a contiguous memory block of several hundred K in the UMB region. Instead, there will be numerous small blocks. So, you should request five 10K blocks instead of one 50K block, as long as your program algorithm will allow it.

DOS function 58H Subfunctions	
Subfunction	Description
00H	Read memory model
01H	Set memory model
02H	Query UMB status
03H	Set UMB status

Memory allocation models

Beginning with Version 2.0, DOS has allowed the use of various memory allocation models. The desired model is selected with subfunction 01H of function 58H. A memory allocation model refers to the way in which DOS searches for a free memory block. Three codes represent the three possibilities listed in the table on the right.

Code	Model
00H	Search low to high
01H	Search for the best fit
02H	Search high to low

Codes 00H and 02H are self-explanatory. These codes allow you to tell DOS to start the search for a memory block of the desired size starting either at the beginning or the end of the RAM. If the first free memory block that's found is at least the requested size, it's allocated.

"Search for the best fit" means that DOS will check the entire RAM for a free memory block that's either exactly the size requested or just a little larger. The idea behind this option is the RAM may be fragmented (e.g., if a TSR program frees some memory before it becomes resident). This leaves holes or "fragments" in the RAM. These fragments are usually very small.

If a program needs a small memory block, you can use this option to search for a memory fragment instead of simply using a small part of the first free memory block DOS finds. Subfunction 01 is used to set the memory allocation model. One of three codes previously listed must be passed in the BL register. In addition to the codes, bit 7 of the BL register is also important in DOS Versions 5.0 and up. This bit can be set or cleared regardless of the code that's used. The status of this bit determines whether the search for the next memory block should begin with the UMBs or in the TPA. To begin with the UMBs, this bit must be set to 1. This is only useful when you've already included UMBs with the proper call to subfunction 03H.

You can query the memory allocation model with subfunction 00H. The result of this function is returned to the AL register. With DOS 5.0 and up, remember to check the value of bit 7.

Viewing Memory Allocation

In this section, we'll discuss a program that enables you to display a graphic representation of RAM allocation on screen. The Pascal and C versions of this program are called MEMDEMOP.PAS and MEMDEMOC.C. Both perform the same task and are almost identical internally.

Both programs allow you to allocate, free and change the size of memory blocks using DOS functions 48H, 49H and 4AH. The location and size of the memory blocks are displayed on screen. Since the screen is too small for the entire TPA and all UMBs, the program is limited to monitoring two specific regions: a 160K region of the TPA and a 40K region from the UMBs.

The program limits itself to these regions by first allocating them as two consecutive blocks at the start of the program. Then the rest of the memory is allocated in 1K blocks so no free 1K blocks remain. We'll explain why 1K is used as the block size shortly.

In the next step, the two large memory blocks are freed again. This means these two regions now contain the only memory that isn't allocated. So, all memory will now be allocated from these regions, as long as 1K or more is requested. The program ensures that all requests for memory are in multiples of 1K.

The **F1** and **F3** function keys can be used to allocate and free memory (see the following illustration). When you press one of these keys, the program will then ask you with which block you want to work. Blocks are labeled within the program as A through Z (with no distinction between upper and lowercase). So, you can choose from a total of 26 blocks.

*Allocating and
freeing memory
with the
programs
MEMDEMOP.P
AS and
MEMDEMOC.C*

```

DOS Memory Management Demo (C) 1991 by Michael Tischer
=====
Memory management strategy: First free memory block used [F8]
First search in UMB       : No [F9]
UMB used                   : No [F10]
-----
Conventional memory:
      1      1      2      3      4
      1      0      0      0      0

```

0K		[F1] Allocate memory [F2] Release memory [F3] Change size [ESC] End program
40K		
80K		
120K		

No UMB available

Enter the desired letter and press **Enter**. Remember the program will accept only those letters that haven't already been used when allocating a memory block. Only the letter of a memory block currently in use will be accepted if you want to free a memory block or change its size.

When you allocate a memory block or change the size of an existing memory block, you'll also be asked to enter the desired size in K. Enter a number and then press **Enter**.

After one of these operations, the program will redraw the screen to reflect the changes that you've made. The memory occupied by an allocated block is identified with the letter that was assigned to the block. Memory that hasn't been allocated appears empty. Each character represents 1K from the TPA or UMB region. This explains the values used for the scale.

The UMB window will appear on screen only if a 40K UMB was allocated at the start of the program. If this window isn't present, either UMBs aren't supported on the system or a 40K block wasn't available.

Memory allocation is interesting to watch when you experiment with the **F1**, **F2** and **F3** function keys. These keys control the way in which memory will be allocated and whether UMBs will be included. They act as toggle switches to switch the various modes on and off.

After switching the modes around, you can see how the next allocated memory block is placed either at the beginning or the end of the available memory in the TPA or UMB region.

To end the program, press the **[Esc]** key. This also frees all memory allocated by the program.

You'll find the following program(s) on the companion CD-ROM

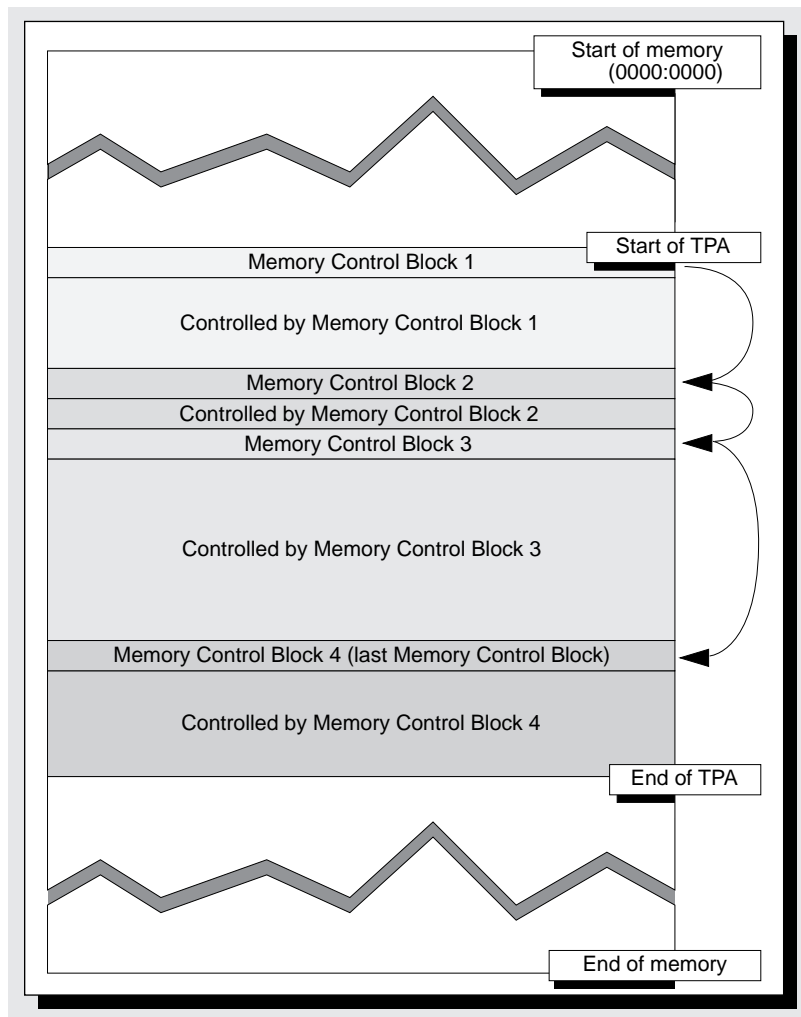


MEMDEMOP.PAS (Pascal listing)
MEMDEMOC.C (C listing)

Behind The Scenes Of Memory Management

The acronym "MCB" in the program listings for the MEMDEMOP.PAS and MEMDEMOC.C programs is an abbreviation for Memory Control Block (MCBs). The MCBs are important in how DOS manages memory.

*Managing
allocated
memory with
Memory Control
Blocks (MCBs)*



To manage memory blocks allocated with DOS function 48H, a Memory Control Block (MCB) is created for each allocated block. An MCB consists of 16 bytes. It always starts at an offset address that is divisible by 16 and it precedes the memory block it describes. The DOS memory management functions always work with the segment address of the allocated block,

so the segment address of the corresponding MCB can be easily obtained by subtracting 1 from the segment address of the allocated block.

Structure of a Memory Control Blocks (MCB) in RAM		
Addr.	Contents	Type
+00h	ID ("Z" = last MCB, "M" = more to follow)	1 BYTE
+01h	Segment address of the corresponding PSP	1 WORD
+03h	Number of paragraphs in the allocated block	1 WORD
+05h	Unused	11 BYTE
+10h	The allocated memory block	x PARAG.
Length: 16 + the size of the allocated block		

The MCB contains three fields, as seen in the previous figure. Mark Zbikowski, one of the developers of MS-DOS, has immortalized himself by using his initials in the first field. If this field contains the letter "M", then additional MCBs follow this one. If it contains "Z", then this is the last MCB in memory.

The second field contains the segment address of the PSP for the corresponding program. This field is only significant if the allocated memory block is for the environment of a program. In this instance, this field establishes a link by pointing to the PSP. Before a program is loaded, the EXEC function allocates a separate memory block for the environment block of the program.

If the memory block is a PSP, then the second field of the MCB usually points directly to the memory block itself.

The third field is more important when evaluating the MCB. This field gives the size of the memory block in paragraphs. Since the next MCB follows directly after this memory block, this number represents the distance to the next MCB minus 1. In this way, each MCB indirectly points to the next one, which generates a linked list of all MCBs.

Accessing the MCB chain

You must know the address of the first MCB to access the entire MCB chain. DOS stores this address in an internal structure called the DOS Information Block (DIB). This structure usually isn't accessible to application programs. However, you can access it with the undocumented function 52H. This function returns a pointer to the DIB in the ES:BX register pair.

Curiously, this address points to the second field of the DIB instead of the first. However, the first field contains the address of the first MCB, which is what we're looking for. The pointer to the first MCB consists of an offset and segment address that occupies four bytes. So, we'll be able to find the desired information at the address ES:[BX-4].

This formula must be used carefully. You cannot simply subtract 4 from the contents of the BX register and end up with the desired address in all cases. This works only if the offset address in the BX register is greater than or equal to 4. If it's smaller, a negative number is the result. As the following example shows, negative numbers aren't used in memory addressing:

If the BX register returns a value of 0 as the offset address of the DIB, then subtracting 4 will result in the value 0FFFCH. In an arithmetic operation, this is correctly interpreted as -4, but as a memory address it points to 0FFFCH, which is at the end of the segment instead of before the given address. The desired information isn't located there.

The solution is to simply decrement the segment address by 1. This reduces the combined segment and offset address by 16. If you now add 12 to the offset address, this results in the original address minus 4, which takes us to the first field of the DIB and the address of the first MCB.

Sample program

- The number of the MCB
- The MCB's address in memory

- Address of the block managed by the MCB
- Contents of the ID field ("M" or "Z")
- Address of the corresponding PSP (regardless of whether it exists)
- Size of the corresponding memory block in paragraphs and bytes

Until DOS Version 3.0, the environment block contained only the previous information. Starting with Version 3.0, the name of the program, to which the environment block belongs, was added after the last environment string. The complete path is given with the program name.

The null byte after the last environment string and the start of the name string are separated by a word. Only if this word contains the value 0001H will the following program name and path be valid. The program name string also ends with a null byte.

An example of program output

To help interpreting the output of this program, we ran the C version on a computer and received the following results (your printout may appear slightly different). Each MCB is explained after the output.

```
MCBC (c) 1988, 91 by Michael Tischer

MCB number      = 1
MCB address     = 09C8:0000
Memory address= 09C9:0000
ID              = M
PSP address     = 0008:0000
Size            = 1554 paragraphs ( 24864 Byte )
Contents        = Unidentifiable as program or data

DUMP | 0123456789ABCDEF          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-----+-----
0000 | n p é! , $CLOCK             6E 01 70 00 08 80 21 00 2C 00 24 43 4C 4F 43 4B
0010 |      é                      20 06 03 80 02 1F 1D 1F 1E 1F 1E 1F 1F 1E 1F 1E
0020 | .æ .ö -PSQR                 1F 2E 89 1E 11 00 2E 8C 06 13 00 CB 50 53 51 52
0030 | WVU ún -> &Å]             57 56 55 1E 06 9C FC 0E 1F C4 3E 11 00 26 8A 5D
0040 | é+ t!é+ tJ +u _           02 80 FB 04 74 21 80 FB 08 74 4A 0A DB 75 03 E9
-----+-----
Please press a key ---

MCB number      = 2
MCB address     = 0FDB:0000
Memory address= 0FDC:0000
ID              = M
PSP address     = 0FDC:0000
Size            = 231 paragraphs ( 3696 Byte )
Contents        = PSP (with program following)
-----+-----
Please press a key ---

MCB number      = 3
MCB address     = 10C3:0000
Memory address= 10C4:0000
ID              = M
PSP address     = 0000:0000
Size            = 3 paragraphs ( 48 Byte )
Contents        = Unidentifiable as program or data
```

```

DUMP | 0123456789ABCDEF      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-----+-----
0000 |      -      -----      00 01 00 00 00 00 00 CB 00 00 00 FF FF FF FF FF
0010 | -----C      FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 43
0020 | : \AUTOEXEC.BAT      3A 5C 41 55 54 4F 45 58 45 43 2E 42 41 54 00 00
0030 | M_      4D DC 0F 0A 00 00 00 00 00 00 00 00 00 00 00 00
0040 | COMSPEC=C:\COMMA      43 4F 4D 53 50 45 43 3D 43 3A 5C 43 4F 4D 4D 41
-----+-----
Please press a key ---

MCB number      = 4
MCB address     = 10C7:0000
Memory address= 10C8:0000
ID              = M
PSP address     = 0FDC:0000
Size            = 10 paragraphs ( 160 Byte )
Contents        = Environment
Program name    = Unknown
Environment string
    COMSPEC=C:\COMMAND.COM
    PATH=C:\;C:\DOS;C:\BATCHES;E:\;D:\MSC\BIN
    INCLUDE=d:\msc\include
    LIB=d:\msc\lib
    TMP=d:\msc\tmp
    PROMPT=[ $p ]

-----+-----
Please press a key ---

MCB number      = 5
MCB address     = 10D2:0000
Memory address= 10D3:0000
ID              = M
PSP address     = 10DD:0000
Size            = 9 paragraphs ( 144 Byte )
Contents        = Environment
Program name    = C:\DOS\KEYB.COM
Environment string
    COMSPEC=C:\COMMAND.COM
    PATH=C:\;C:\DOS;C:\BATCHES;E:\;D:\MSC\BIN
    INCLUDE=d:\msc\include
    LIB=d:\msc\lib
    TMP=d:\msc\tmp

-----+-----
Please press a key ---

MCB number      = 6
MCB address     = 10DC:0000
Memory address= 10DD:0000
ID              = M
PSP address     = 10DD:0000
Size            = 341 paragraphs ( 5456 Byte )
Contents        = PSP (with program following)

-----+-----
Please press a key ---

MCB number      = 7
MCB address     = 1232:0000
Memory address= 1233:0000
ID              = M
PSP address     = 123D:0000
Size            = 9 paragraphs ( 144 Byte )

```

```

Contents      = Environment
Program name  = C:\DOS\CED.COM
Environment string
               COMSPEC=C:\COMMAND.COM
               PATH=C:\;C:\DOS;C:\BATCHES;E:\;D:\MSC\BIN
               INCLUDE=d:\msc\include
               LIB=d:\msc\lib
               TMP=d:\msc\tmp

----- Please press a key ---
MCB number    = 8
MCB address   = 123C:0000
Memory address= 123D:0000
ID            = M
PSP address   = 123D:0000
Size          = 1030 paragraphs ( 16480 Byte )
Contents      = PSP (with program following)

----- Please press a key ---
MCB number    = 9
MCB address   = 1643:0000
Memory address= 1644:0000
ID            = M
PSP address   = 164E:0000
Size          = 9 paragraphs ( 144 Byte )
Contents      = Environment
Program name  = C:\DOS\CACHE-AT.COM
Environment string
               COMSPEC=C:\COMMAND.COM
               PATH=C:\;C:\DOS;C:\BATCHES;E:\;D:\MSC\BIN
               INCLUDE=d:\msc\include
               LIB=d:\msc\lib
               TMP=d:\msc\tmp

----- Please press a key ---
MCB number    = 10
MCB address   = 164D:0000
Memory address= 164E:0000
ID            = M
PSP address   = 164E:0000
Size          = 1922 paragraphs ( 30752 Byte )
Contents      = PSP (with program following)

----- Please press a key ---
MCB number    = 11
MCB address   = 1DD0:0000
Memory address= 1DD1:0000
ID            = M
PSP address   = 1DDC:0000
Size          = 10 paragraphs ( 160 Byte )
Contents      = Environment
Program name  = C:\DOS\KEYBUF.COM
Environment string
               COMSPEC=C:\COMMAND.COM
               PATH=C:\;C:\DOS;C:\BATCHES;E:\;D:\MSC\BIN
               INCLUDE=d:\msc\include
               LIB=d:\msc\lib

```

```

                TMP=d:\msc\tmp
                PROMPT=[ $p]

----- Please press a key -----
MCB number      = 12
MCB address     = 1DDB:0000
Memory address= 1DDC:0000
ID              = M
PSP address     = 1DDC:0000
Size            = 27 paragraphs ( 432 Byte )
Contents        = Unidentifiable as program or data

DUMP | 0123456789ABCDEF          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-----+-----
0000 |  M M M M M M M M          00 4D 00 4D 00 4D 00 4D 00 4D 00 4D 00 4D 00 4D
0010 |  M M M M M M M M          00 4D 00 4D 00 4D 00 4D 00 4D 00 4D 00 4D 00 4D
0020 |  + 1  K K K K K K K K      2B 1B 31 02 00 4B 00 4B 00 4B 00 4B 00 4B 00 4B
0030 |  K K K K K K K K          00 4B 00 4B 00 4B 00 4B 00 4B 00 4B 00 4B 00 4B
0040 |  K K K K K K K K          00 4B 00 4B 00 4B 00 4B 00 4B 00 4B 00 4B 00 4B
-----+-----
----- Please press a key -----
MCB number      = 13
MCB address     = 1DF7:0000
Memory address= 1DF8:0000
ID              = M
PSP address     = 0FDC:0000
Size            = 4 paragraphs ( 64 Byte )
Contents        = PSP (with program following)

----- Please press a key -----
MCB number      = 14
MCB address     = 1DFC:0000
Memory address= 1DFD:0000
ID              = M
PSP address     = 1E08:0000
Size            = 10 paragraphs ( 160 Byte )
Contents        = Environment
Program name    = D:\PCI\C\TC.EXE
Environment string
                COMSPEC=C:\COMMAND.COM
                PATH=C:\;C:\DOS;C:\BATCHES;E:\;D:\MSC\BIN
                INCLUDE=d:\msc\include
                LIB=d:\msc\lib
                TMP=d:\msc\tmp
                PROMPT=[ $p]

----- Please press a key -----
MCB number      = 15
MCB address     = 1E07:0000
Memory address= 1E08:0000
ID              = M
PSP address     = 1E08:0000
Size            = 16200 paragraphs ( 259200 Byte )
Contents        = PSP (with program following)

----- Please press a key -----
MCB number      = 16
MCB address     = 5D50:0000

```



```

Memory address= 5D51:0000
ID              = M
PSP address     = 5D5C:0000
Size           = 10 paragraphs ( 160 Byte )
Contents       = Environment
Program name    = C:\TC\OBEX\MCBC.EXE
Environment string
                COMSPEC=C:\COMMAND.COM
                PATH=C:\;C:\DOS;C:\BATCHES;E:\;D:\MSC\BIN
                INCLUDE=d:\msc\include
                LIB=d:\msc\lib
                TMP=d:\msc\tmp
                PROMPT=[ $p ]

----- Please press a key ---
MCB number      = 17
MCB address     = 5D5B:0000
Memory address= 5D5C:0000
ID              = M
PSP address     = 5D5C:0000
Size           = 4512 paragraphs ( 72192 Byte )
Contents       = PSP (with program following)

----- Please press a key ---
MCB number      = 18
MCB address     = 6EFC:0000
Memory address= 6EFD:0000
ID              = Z
PSP address     = 0000:0000
Size           = 12547 paragraphs ( 200752 Byte )
Contents       = Unidentifiable as program or data

DUMP | 0123456789ABCDEF          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-----+-----
0000 | H   6   % É   +           48 00 00 00 36 00 08 00 25 00 88 08 00 00 A8 00
0010 | (   v   --P   '           28 04 1E 00 76 00 17 00 FF FF 96 08 00 00 27 0C
0020 | H   Q   --P â   .           48 1F 00 00 51 00 1E 00 FF FF 96 08 90 01 2E 05
0030 | HB   6   % P _   B           48 42 00 00 36 00 08 00 25 00 96 08 EC 01 42 00
0040 | ( | v   --û   E           28 0D E3 00 76 00 17 00 FF FF A4 08 00 00 45 0C
----- Please press a key ---

```

- 1 Although the program couldn't identify the first MCB (so the given PSP address isn't significant), the memory dump provides information about its contents. The first line of the ASCII dump contains the word "\$CLOCK", which is the name of the DOS device driver for the internal clock.

As you might expect, this does look like a device driver. The first 18 bytes corresponds to the exact structure of a device driver header. However, this cannot be a permanently installed device driver from DOS because these are installed below the TPA (Transient Program Area) and don't require memory to be allocated for them.

So, this must be a driver that's installed with the DEVICE command from within the CONFIG.SYS file. The first device driver installed in our CONFIG.SYS file is "AT-UHR.SYS". This device uses the name "\$CLOCK" as its device name.

This driver requires only a few kilobytes, but the allocated memory block is much larger than this. So, there must be more program code or data after this driver. By examining the five lines of the dump, we can see that all drivers that were installed with the DEVICE command can be found here. This means the first memory is already allocated by DOS

during the boot procedure. It's given to all device drivers in the order in which they are named in the CONFIG.SYS file.

- 2 This memory block apparently contains a program. Since it's not preceded by an environment block that would provide the name of the program, we don't know which program this is. But from its location in the MCB chain and in RAM, we know that it was installed as a resident program shortly after the system was booted.
- 3 The contents of this memory block don't provide much information. Either it was allocated by a program for storing data at a later time, or it was simply left over after a memory block was freed.
- 4 This is obviously an environment block, but the corresponding program name is missing. 0FDC:0000 is given as the PSP address, which corresponds to the memory block managed by MCB 2. Since MCB 2 is a PSP and MCB 4 is an environment block, we can be almost certain these two blocks represent a program and its environment. Since the environment block doesn't have a program name associated with it, we can also conclude that this program wasn't started from the DOS command line or by a command in a BATCH file.

MCB 2 seems to represent the resident portion of the command processor COMMAND.COM, with the environment managed by MCB 4. This is confirmed by examining the program code in MCB 2 with a debugger.
- 5 This is the environment block for the program KEYB.COM, which enables us to work with the German keyboard. This program is started within our AUTOEXEC.BAT file using the command KEYB.GR. This is a resident program that remains in memory after it's installed.
- 6 The environment for KEYB.COM is in MCB 5 and the actual program is in MCB 6 (the PSP, meaning the program code and data). This is because the PSP address given in MCB 5 points to the memory block managed by MCB 6.
- 7, 8 These two blocks represent the environment and PSP for the CED.COM program, which is also started from within AUTOEXEC.BAT. This is also a memory resident program.
- 9, 10 Same as MCBs 7 and 8, except it's for the program CACHE-AT.COM.
- 11, 12 These blocks are for the KEYBUF.COM program. The environment block is clearly present, but the other block cannot be readily identified as the PSP. This is because this program uses (or rather misuses) a keyboard buffer as its PSP. So, the interrupt call at the start of the PSP is overwritten. This command identifies the PSP; without it the block cannot be identified as a block.
- 13 The program indicates that this block is occupied by a PSP. However, it must be only the beginning of a PSP, since this memory block is only 64 bytes and a PSP needs 256 bytes. This block was probably occupied by a PSP that wasn't completely overwritten after it was freed. So some of it still remains in memory.
- 14, 15 The program for outputting MCB contents was written in C, so we were in the Turbo C environment when we ran it. MCBs 14 and 15 therefore contain the environment and program code for Turbo C.
- 16, 17 To create this MCB memory dump, we compiled, linked, and executed the program MCBC within Turbo C. Executing this program also creates another process that starts Turbo C with the help of the EXEC loader. So, the block managed by MCBs 16 and 17 were allocated by the EXEC loader to run the program. After the program ends, they will be freed again.
- 18 The last memory block contains all the remaining memory that wasn't allocated at the time. This is about 200K of memory.

The MCBP.PAS, MCBC.C or MCBB.BAS demonstration program can be used to generate MCB dumps like the one we've described. The three versions are very similar. The BASIC version is slightly different because BASIC cannot use FAR pointers to query the memory. Instead, the PEEK and DEF SEG commands must be used (see Chapter 2 for more information).

You'll find the following program(s) on the companion CD-ROM



MCBB.BAS (BASIC listing)
MCBP.PAS (Pascal listing)
MCBC.C (C listing)



The EXEC Function

We've briefly mentioned the EXEC function when discussing the command processor. We'll examine the EXEC function closely and describe how it operates in this chapter.

Parent/child

The EXEC function is one of the many DOS functions that can be called with interrupt 21H (function 4BH). This function allows a parent program (main program) to call a child program (secondary program). The child program is loaded from a mass storage device into memory and then executes. If this child program doesn't become resident, the memory occupied by the child is released following program execution. The child program can also call another program that works with the parent program. This creates a type of program chaining that's limited only by the amount of available RAM.

One example of the EXEC function is the command processor. Using the EXEC function, the command processor executes user-specified programs and becomes the parent program. Some programs (such as Microsoft Word) permit the user to execute DOS commands from the main program using this function. The parent program can pass parameters to the child program in the command line and can also pass parameters using the environment block. This program can also transfer information to the child program within the PSP. Since, like all executable programs, a PSP precedes the child program, information can be entered into the two FCBs within this PSP and made accessible to the child program.

Child program

When control is transferred to the child program, this program can access all the files and devices previously opened by the parent program (or one of the parent programs) with a handle function. This allows the child program to read information from a file or write information to a file whose handle is known (the child program doesn't need to know the filename). This is only possible if the handle was passed by the parent program in one of the three methods we described or if the child program refers to one of the five handles, which are always open. These file accesses affect the file pointer. Since values aren't reset, these file accesses become -visible+ to the parent program when control returns to the parent program.

After the child program executes, control returns to the parent program and execution continues. To pass information (e.g., an error that occurred during the execution of the child program), the child program can pass a numeric value at the end of its execution. This can be done using DOS function 4CH, which terminates a program and returns a code to the parent program.

The communication between the parent and child programs works only if both programs agree on this return value. After control returns to the parent program, it can determine the code using function 4DH of interrupt 21H. To use function 4DH, only the function number is passed in the AH register. The code passed by the child program is returned to the calling (parent) program in the AL register.

Ending the child program

The contents of the AH register indicate how the child program terminated. The value 0 indicates a normal termination, while the value 1 indicates the child program terminated when the user pressed **Ctrl** **C** or **Ctrl** **Break**. If an error during access to a mass storage device forced the child program to terminate, a code of 2 is passed in the AH register. The value 3 indicates the child program terminated from a call to function 31H, or interrupt 27H; the child program then becomes resident in memory.

As we mentioned, the EXEC function can load the child program only if sufficient memory is available. While DOS can estimate the memory needed for EXE programs fairly accurately, it cannot do the same for COM programs. For COM programs, DOS reserves all unused memory. Because of this, a COM program cannot call another program with the EXEC

function because DOS doesn't reserve any extra memory. This also applies to many EXE programs. If a call to a child program is necessary, the required memory space must be released from the calling program before the EXEC function is called (refer to the "Using BIOS To Access The Hard Drives" section in Chapter 14 for explanations on how to do this).

EXEC

If the EXEC function is called, the various parameters are loaded into the registers before calling interrupt 21H. Function number 4BH is passed in the AH register. A value of 0 or 3 is passed in the AL register. A value of 0 indicates the EXEC function will load and execute the program, while a value of 3 indicates the program is loaded as an overlay (without executing it). The address of the name of the program to be loaded or executed is passed in the DS:DX register pair and the address of the parameter block is passed in the ES:BX register pair.

The program name is specified as an ASCII string and ends with a null character (ASCII code 0). This name can include the device name and a complete path description. Its last element is the program name that, besides the name itself, must have either the .COM or .EXE extension. If the device name or path designation are omitted, the system searches for the program in the current directory of the current device. Since the EXEC function cannot execute a batch file directly, the program name that's passed cannot contain the .BAT extension.

Batch child

If a batch file must be executed, first the COMMAND.COM (command processor) file must be invoked. To indicate that a batch file should be executed, the parameter /c, followed by the name of the appropriate batch file, is added to the command line. Calling the command processor with the /c parameter also enables you to call any other program and even internal DOS commands, such as DIR.

Besides directly calling a program, it's also possible to specify program names without file extensions during a command processor call. The command processor searches for an EXE file, then a COM file, and finally a BAT file. If none of these files exist in the current directory, it searches all directories specified in the PATH command. This procedure isn't used during a direct program call without the addition of the command processor. The directory that contains the command processor should be specified. If it isn't specified, it will be loaded from the path indicated by the COMSPEC environment string of the SET command.

Parameter blocks

Parameters can be passed to the command processor in the parameter block following the program name. These are the same parameters that are entered from the keyboard when the program is called. Later we'll see how these parameters affect the EXEC function. However, first we must discuss the parameter block's structure when the AL register contains the value 0. The address for this block is passed to the EXEC function in the register pair ES:BX.

1	0- 1	Segment address: Environment block
2	2- 3	Offset address: Command parameter
3	4- 5	Segment address: Command parameter
4	6- 7	Offset address: First FCB
5	8- 9	Segment address: First FCB
6	10-11	Offset address: Second FCB
7	12-13	Segment address: Second FCB

Field 1 indicates the segment address of the child program's environment block. This block doesn't require an offset address because it always starts at a location divisible by 16. So, its offset address is always 0.

Environment block

The command processor and other programs obtain information from the environment block. This is a series of ASCII character strings. This information can include paths for file searches. Each string has the following syntax, which ends in a null character (ASCII code 0):

```
Name = Parameter
```

The individual strings follow each other sequentially (i.e., the null character of one string is immediately followed by the first character of the next string). Environment blocks can have a maximum length of 32K. The user can change the environment

block by using the DOS SET and PATH commands. Programs that remain resident after execution are unaffected by any changes made to the environment block through these two DOS commands.

If the parent program wants to pass information to the child program using the environment block, it can either construct a new environment block or add the appropriate information to its own environment block. In the first instance, the segment address of the new environment block is specified in the first field of the parameter block. If the child program should have access to the environment block of the parent program, specify a value of 0 in this field. Before turning over control to the child program, the EXEC function stores the segment address of the environment block in the memory location at address 2CH of the child program's PSP. If the child program must use a new environment block, it should contain at least 3 strings that are usually part of the environment block of the parent program and are important to the command processor:

```
COMSPEC = Parameter
PATH = Parameter
PROMPT = Parameter
```

If a child program modifies its environment block, the parent program's environment block remains unchanged after the child program completes its execution. Fields 2 and 3 indicate the command parameters' address that is passed to the PSP of the program starting at address 80H. These fields must have the same structure in memory as expected by DOS in the PSP. The first byte indicates the number of command characters minus 1. This is followed by the command characters as normal ASCII codes. The command parameters terminate with a carriage return (ASCII code 13), which isn't included in the character count. For compatibility with COMMAND.COM, the first character in the string should be a space.

To call a batch program (called DO.BAT) using the command processor, the following command parameters must be specified as a string in memory:

```
DB 10, " /C DO.BAT", 13
```

The EXEC function copies the command parameters into the PSP of the program to be executed. It also removes all the parameters that would redirect the input or output, because a redirection of the standard input/output can only be performed by the parent program. The child program can still use input/output redirection if the standard input/output handles have been redirected by the parent program. (See Chapter 19 for more information and an example of this process.)

Fields 6, 7, 10, and 11 indicate two FCBs installed in the PSP at address 5CH or 6CH. If this isn't required, specify -1 (FFFFH) in these two fields. However, if this is needed for program execution, enter the first two command parameters in the two FCBs with DOS function 29H. Before passing control to the child program, the EXEC function copies these two FCBs into the PSP of the child program.

Although all the registers and the parameter block now have the required values, the EXEC function cannot be called yet. Since this function destroys the contents of all registers up to the CS and IP registers during execution, the contents of all the registers must be placed on the stack before this function is activated. Then the contents of the SS and SP registers must be stored within the code segment. Only then can interrupt 21H function 4BH be called to activate the EXEC function. After the EXEC function ends, the carry flag indicates whether the function executed normally. Before program execution can continue, the value of the SS and SP registers must be restored from the code segment. Then the contents of the other register can be restored again from the stack.

The EXEC function performs a different task when a value of 3 appears in the AL register. In this case, it loads a COM program or an EXE program into memory without executing. After the target program is loaded, control immediately returns to the calling program.

Unlike subfunction 0, the program loads to a memory address indicated by the calling program instead of loading to any nonspecific location. Since parameters aren't passed to the loaded program, the parameter block has a different structure during the call of subfunction 3 than during the call of subfunction 0:

Field	Byte	Purpose
1	0- 1	Segment address where overlay is located
2	2- 3	Relocation factor

Before the function is called, the segment address to which the program should be loaded is specified in the first field of the parameter block. If the calling program doesn't have enough memory available for loading the external program, it should request additional memory with one of the DOS memory management functions. The loaded program loads directly to the segment address indicated with the offset address 0 because a PSP doesn't precede the program.

Relocation

The relocation factor adjusts the segment address of the called program. Since this factor applies only to EXE programs (COM programs cannot have specific segment assignments), the relocation factor for COM programs should always equal 0. The relocation factor for EXE programs should indicate the segment address where the program will be loaded to conform to the program's segment assignments. After the program is loaded, its routines are ready to be accessed. The routines of the loaded program should always be treated as subroutines (i.e., called with the machine language CALL instruction). It must always be a FAR type instruction although the loaded program may be located immediately after the calling program, but it can never have the same segment address.

The offset address for CALL is always 100H for a COM program because execution always starts immediately after the PSP at address 100H. However, this creates a problem. Subfunction 3 prevents the PSP from loading. So, the code segment of the COM program starts at address 0, instead of at the offset address 100H (relative to the load segment). Since all jump instructions and accesses to data within the COM program are relative to address 100H instead of address 0, you cannot execute a FAR CALL instruction with the address of the load segment as the segment address, and address 0 as the offset address. The segment address for the FAR CALL must indicate the address of the load segment minus 10H and the address 100H as the offset address. If the COM program specifically acts as an overlay for another program, entry addresses other than address 100H are possible. In this case, only the offset address for the FAR CALL instruction changes. The segment address must remain 10H smaller than the address of the load segment.

EXEC and memory

The problem is different for EXE programs. If these programs are loaded for execution using subfunction 0, the EXEC function sets the code segment and the instruction pointer to the instruction that was declared as the first instruction in the assembler source. However, this address is unknown to the program that loaded the EXE program as an overlay. This problem can easily be solved by placing the first executable instruction in the EXE program at the beginning of the EXE program. This makes its offset address 0. The EXE program source must not be in the normal sequence with the stack first. In this case, the code segment must be the first segment in the source to ensure that it begins the EXE program. The FAR CALL uses the address of the load segment as the segment address, and address 0 as the offset address.

Demonstration program

While BASIC, Pascal and C have commands or procedures to call a program from another program, assembly language routines must use DOS function 4BH. The EXEC.ASM program which you'll find on the companion CD-ROM is an example program that should help you understand this function.

The framework of the EXE program listed in Chapter 19 acts as the basis for this program. The EXEPRG procedure performs the actual work. This procedure calls the new program using function 4BH. Two strings, which contain the name of the program to be called and the necessary parameters, are passed to it. Both strings end with the null character (ASCII code 0). All the variables EXEPRG needs for execution are located in the code segment. The advantage of this method is that, to use this routine, the lines from the code segment must be copied into only one of the application programs. After calling EXEPRG, the carry flag signals whether an error occurred. If true (carry flag=1), the AX register contains the error code as returned by the EXEC function of DOS. If the called program executed correctly, the carry flag is reset (0) and the termination code of the called program, as returned by DOS function 4DH, is returned by the AX register.

Within this program, EXEPRG displays the current directory using the command processor. The command processor defaults to the current directory of the current device.

You'll find the following program(s) on the companion CD-ROM



EXEC.ASM (Assembler listing)

26

Ctrl Break And Critical Error Interrupts

Two ways are possible in DOS to stop a program during execution. This occurs when the user presses **Ctrl Break** (**Ctrl C**) or when a critical error occurs during access to an external device (i.e., printer, hard drive, disk drive, etc.). Although the key combination that's used depends on the PC configuration, we'll use **Ctrl Break** in this section.

Pressing **Ctrl Break** to stop a program during execution can have serious consequences. After the user presses this key combination, DOS abruptly takes control from the program without allowing the program to perform any necessary "housekeeping". For example, files aren't closed properly, diverted interrupt vectors aren't reset, and allocated memory isn't released. This can result in data loss or a system crash. To prevent this from happening, DOS calls interrupt 23H, which is also known as the **<Ctrl><Break>** interrupt. When a program is started, this interrupt points to a routine that causes the program to end. However, a program is able to select a routine of its own, which enables the program to maintain control of what occurs when the user presses **Ctrl Break**.

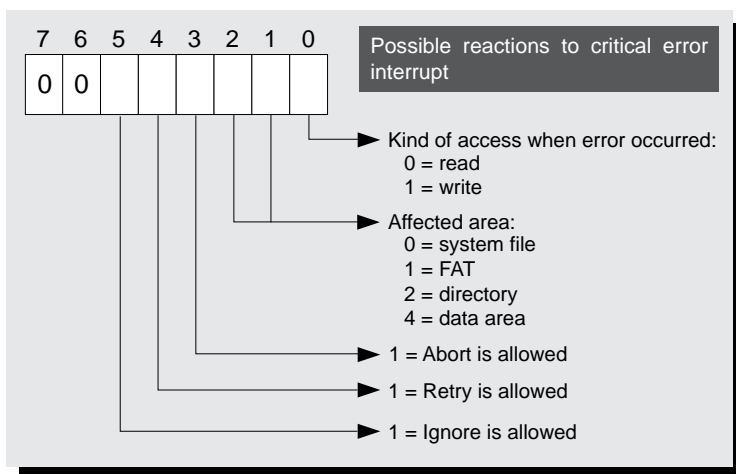
However, the interrupt routine doesn't execute immediately. Instead, the break flag determines when this routine occurs. This flag can be set at the DOS prompt using the **BREAK (ON/OFF)** command from DOS, or with the help of DOS function 33H, subfunction 1. If the break flag is on, every time a function of DOS interrupt 21H is called, the keyboard buffer will be checked to see if either **Ctrl Break** or **Ctrl C** has been pressed. If the break flag is off, this check will be performed only when calling the DOS functions that access the standard input and output devices.

If this test finds the appropriate key combination, the processor registers are loaded with the values contained in the DOS function to be executed. Interrupt 23H is called only after this occurs. If a program directs this interrupt to a routine of its own, several things may occur. For example, the program could display a window on the screen that asks whether the user wants to end the program. The program can also decide for itself whether the program should end.

Maintenance

If the program chooses to stop execution, some type of clean-up routine should be performed. This type of routine closes all open files, resets any changed interrupt pointers, and releases any allocated memory. After this, function 4CH can end the program without returning control to the interrupt 23H caller.

The **IRET** assembly language instruction must return control to DOS if **Ctrl Break** should be ignored. The program must then ensure that all processor registers contain the same values they had when interrupt 23H was invoked. Otherwise, the DOS function that was originally called cannot be performed without an error.



We'll demonstrate both of these methods in an example at the end of this section.

Critical error interrupt

Unlike the **Ctrl Break** interrupt, the critical error interrupt call usually isn't a reaction to something the user does intentionally. Instead, it's usually a reaction to an error that occurs when accessing an external device, such as a printer, disk drive, or hard

disk. Although the user can correct the error in many cases (e.g., the printer isn't switched on), other errors can be caused by hardware failures that require repairs (e.g., read error while accessing the hard drive).

To make allowances for the various kinds of errors, the critical error interrupt (interrupt 24H) usually points to a DOS routine that displays the following or a similar message on the screen and waits for input from the user:

```
(A)bort (R)etry (I)gnore (F)ail
```

This clears the currently executing program from the screen. Similar to **Ctrl Break**, this interrupt ends the program abruptly. So, the files aren't properly closed, allocated memory isn't released, etc. Installing an interrupt handler in a program to replace the DOS handler can help. DOS uses a processor register to pass this handler various information when it's called. This helps the interrupt handler locate the source of the error. Bit 7 in the AH register indicates either a floppy or hard disk access error (bit 7 off) or some other error (bit 7 on). The other bits give information about the reaction to possible error codes.

Also, the BP:SI register pair points to the head of the device driver that was being called when the error appeared. A detailed error code is contained in the lower 8 bits of the DI register and the contents of the upper 8 bits are undefined. This returns the error codes shown in the table on the right.

Error codes passed to the critical error handler			
Code	Meaning	Code	Meaning
00h	Disk is write protected	07h	Unknown device type
01h	Access to an unknown device	08h	Sector not found
02h	Drive not ready	09h	Printer out of paper
03h	Unknown command	0Ah	Write error
04h	CRC error	0Bh	Read error
05h	Wrong data length	0Ch	General error
06h	Seek error		

When called, the critical error handler can respond by opening a window on the screen that asks the user to decide to ignore the error, retry the access, or abort the program. The last option can only instruct the interrupt to call DOS functions 01H to 0CH. This means the program ends abruptly, similar to pressing **Ctrl Break**.

Although calling other DOS functions within the handler doesn't cause errors itself, the return to DOS causes a system crash. These handlers also aren't allowed to end a program by using DOS function 4CH. Instead, the handler must return to its caller with the help of the IRET command. With that, DOS expects a code in the AL register that will show it how to react to the error. It interprets the contents of the AL register as according to the table on the right.

Output codes of a critical error handler	
Code	Meaning
00h	Ignore error
01h	Retry operation
02h	End program with interrupt 23H
03h	End function called with error (DOS 3.0 and up)

The last output code in the previous list represents the most sensible reaction to an error that can't be fixed by repeating the operation (e.g., when the printer must be switched on). The receipt of this code invokes the normal ending of the function call, in which the error occurred. The function then sets the carry flag to signal the error. While this makes a "critical" error and a "normal" error indistinguishable to the program, it's possible to distinguish them by setting a flag within the critical error handler.

You'll find the following program(s) on the companion CD-ROM



CEBHAND.ASM
(Assembler listing)

27

Device Managers

The device driver is one of the most fascinating and most complicated, aspects of system programming. DOS uses device drivers to access external devices. Device drivers are short programs that provide support to a wide variety of devices, ranging from keyboards to CD-ROM drives. However, they are difficult to use because they can be programmed only in assembly language. In this chapter, we'll show you how device drivers are structured under DOS. You'll also learn how to develop your own drivers. We've included source code for several functional drivers, which can also serve as the basis for your own, more complicated drivers.

Device Drivers Under DOS

A device driver is the part of the operating system that's responsible for controlling and communicating with the hardware. It represents the lowest level of an operating system and permits all other levels to work independently of the hardware. This is useful when adapting an operating system to various computers because the device drivers can be changed instead of the entire operating system. In earlier operating systems, device drivers resided in the operating system code. This meant that changing or upgrading these routines to match new hardware was difficult, if not impossible. DOS Version 2.0 introduced a flexible concept of device drivers. This flexibility allows user to adapt even the most exotic hard drives and EMS expansion cards to DOS.

A device driver consists of status information that tells DOS what kind of driver it is and several software routines known as driver functions. These routines are responsible for tasks required by DOS to access the device the driver serves. For example, a hard drive device driver must contain functions for handling read, write and verify operations on sectors of the disk.

Custom drivers

Since communication between DOS and a device driver is based on relatively simple function calls and data structures, the assembly language programmer can develop a device driver to adapt any device to DOS. Unfortunately, device drivers cannot be programmed in a higher level language, as we mentioned earlier.

The rules for developing a COM program also apply when developing the code for a driver. Direct segment access isn't allowed. The difference is that a device driver starts at offset address 0H, instead of 100H. The end of this section explains the assembly language implementation in detail.

Device drivers are installed by DOS during the boot process. They cannot be activated from the command line like normal EXE or COM programs. The drivers present in the DOS kernel are installed automatically during the boot process. These drivers are named \$CLOCK device driver, CON device driver, AUX device driver and PRN device driver. In addition to these, drivers for the diskette drives and the hard disk will also be installed, if available.

Name	Driver for
NUL	Null (imaginary) device
\$CLOCK	Clock
CON	Console (keyboard and screen)
AUX	Serial port
PRN	Parallel port (printer)

The drivers are arranged sequentially in memory and connected to each other. If the user wants to install another driver, DOS must be informed using the CONFIG.SYS file. This text file contains the information that DOS needs for configuring the system. Contents of the CONFIG.SYS file are read and evaluated during the boot process after linking the standard drivers. If DOS finds the DEVICE= command, the driver described in that line is installed, based on the optional path.

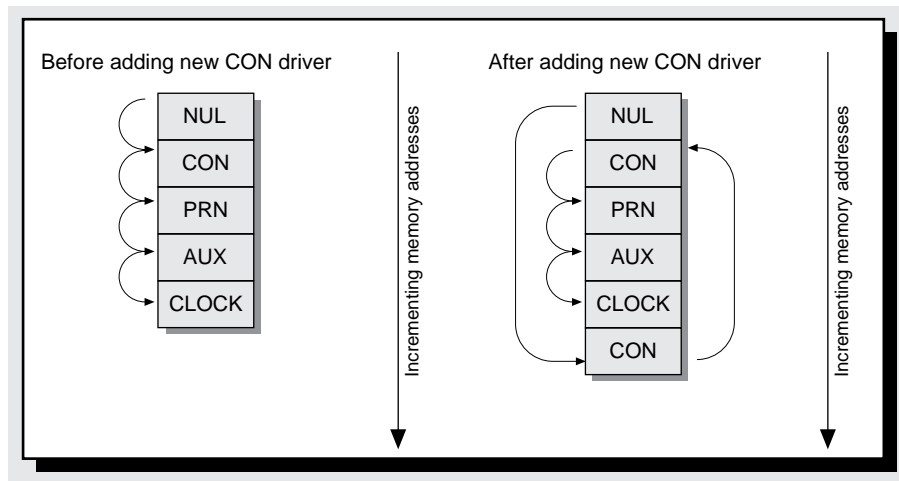
ANSI.SYS

The following command sequence includes the ANSI.SYS driver, which is supplied with DOS. This driver makes enhanced character output and keyboard functions available:

```
DEVICE=ANSI.SYS
```

The new driver is added to the chain immediately after the NUL device driver (the first driver in the chain). The ANSI.SYS driver replaces the default CON driver. To ensure that all function calls for monitor or keyboard communication operate through ANSI.SYS, the ANSI.SYS driver is placed first in the device group and the CON driver is moved farther down the chain of devices. Since the operating system moves from link to link during the search, it finds the new CON driver (ANSI.SYS) first and uses it. So, the system ignores the old CON driver, as shown in the following illustration:

The driver chain before and after adding new CON driver



ASSIGN

Not all drivers can be replaced with new ones. The NUL driver is always the first driver in the chain. If you add a new NUL driver, the system ignores this driver and continues accessing the original NUL driver. This also applies to the drivers for floppy disk drives and hard drives. This occurs because disk drives have drive specifiers instead of names such as CON (e.g., A:). A new disk drive can be added to the system, but since DOS may assign it the name D:, it may not be addressed by all programs that want to access device A:. Avoid this problem by redirecting all device accesses using DOS's ASSIGN command. You can make the ASSIGN command part of the AUTOEXEC.BAT file. It executes after adding drivers and executing the CONFIG.SYS file. To redirect all accesses from drive A: (the first disk drive) to device D: (in this case, a new driver for a new disk drive), the AUTOEXEC.BAT file must contain the following command sequence:

```
ASSIGN A=D
```

The drivers for mass storage devices and the drivers, such as PRN, are handled differently. The two types of device drivers used by DOS are character device drivers and block device drivers.

Character device drivers

Let's start with character device drivers because they have a simple structure. Character device drivers transmit one byte for every function call. They communicate with devices, such as the keyboard, display, printer and modem. Since a device driver can service only one device, individual drivers for keyboard, display, printer, etc., exist in DOS after booting. Character devices can operate in either cooked mode or raw mode.

Cooked mode and raw mode

In cooked mode, the device driver reads characters from the device and performs a test for certain control characters. DOS then passes the character to an internal buffer. DOS also checks to determine whether any **Enter**, **Ctrl P**, **Ctrl S** or **Ctrl C** characters exist. If the system detects the **Enter** character, it ignores any further input from the device driver, even if the specified number of characters hasn't been read yet. Then the characters read are copied from the internal buffer to the buffer of the calling program. If characters are output in cooked mode, DOS tests for **Ctrl C** or **Ctrl Break**. If one of these combinations is detected, the currently running program stops. Pressing **Ctrl S** temporarily stops the program until the user presses any other key. **Ctrl P** redirects the output from the screen to the printer (PRN). Pressing **Ctrl P** a second time redirects the output from the printer back to the screen.

In raw mode, the device driver reads all characters without testing. If a program wants to read in 10 characters, it reads exactly 10 characters, even if the user presses the **Enter** key as the second character of the string. Raw mode transmits the characters directly to the calling program's buffer, instead of using an internal DOS buffer. During character output, raw mode doesn't test for **Ctrl C** or **Ctrl Break**. DOS function 44H of interrupt 21H defines the mode of the character device driver (see the end of this section for a detailed description of this interrupt).

Block device drivers

Block device drivers usually communicate with mass storage devices, such as hard drives. Therefore, they simultaneously transmit a number of characters designated as a block. In some cases, a single call to a function transmits several blocks of data. The block sizes can differ depending on the mass storage device and within one particular mass storage device.

When DOS wants to access a storage medium with a block driver, DOS passes the number of the sector being addressed. The driver must then convert the logical sector number (counted starting from 0) to a physical address consisting of head, cylinder and sector numbers. The device driver selects the method used for converting logical sector numbers into physical sector addresses. It's only important that a unique, one-to-one relationship is maintained between logical and physical sectors. This means that a given logical sector uniquely identifies one and only one physical sector.

Unlike character device drivers, block device drivers can manage more than one device at a time. A single hard disk driver can work with two or more hard disks at once, for example. A block driver can also divide a mass storage device into several volumes, which is often done with hard drives.

Identifying devices managed by a driver

The drives managed by a block driver don't have device names or filenames. Instead, they use identifying letters such as A, B or C. The device letters are assigned by DOS and aren't selected by the driver. The letters DOS assigns are determined by the location of the block driver within the list of drivers. The first drive managed by a block driver receives the letter A, the second B and then C, etc. Each device must have a file allocation table (FAT) and a root directory. Block device drivers don't distinguish between cooked and raw modes. They always read and write the exact number of blocks unless an error is detected.

If a device driver supports several logical drives, these are assigned consecutive letters. For example, if a hard disk driver implements three logical drives and the first letter available is C, then the other two drives are assigned the letters D and E. The next block driver then begins with the letter F.

Device driver access

Several ways are available to access a device driver. Character device drivers are accessed using the normal FCB or handle functions by simply indicating the name of a driver (e.g., CON: instead of a filename). A block device driver is accessed using the normal DOS functions (file, directory, etc.) by using the drive designator assigned by DOS during the boot process.

Functions 1H through CH of interrupt 21H invoke read and write operations in a device driver. There are also two other options for accessing device drivers, which we'll discuss shortly. This can no longer be considered direct access, since various DOS functions are used to communicate between the driver and the device. Two other ways to communicate with device drivers are possible. DOS function 44H, which is known as the IOCTL function (I/O control), is important to this communication. This function has many subfunctions that we'll discuss in later.

Structure Of A Device Driver

Although the two types of device drivers differ in some important ways, they have similar structures. Each has a device header, a strategy routine and an interrupt routine (a different kind of interrupt from the ones we've discussed so far).

Device header

The device header appears at the beginning of each device driver and contains information that DOS needs for implementing the driver.

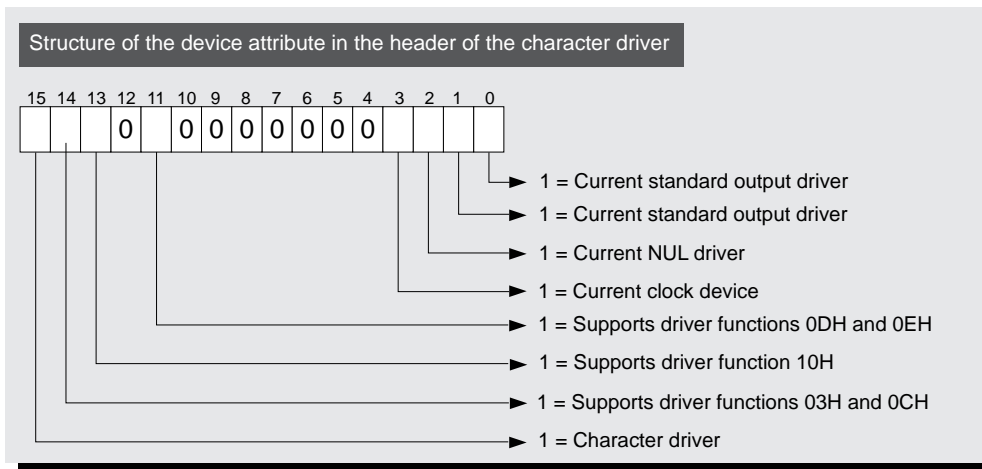
Device driver header

The first field creates a link to the next device driver in the list. Once a driver is loaded, DOS enters the address of the next driver in this field. The programmer must initialize this field with the value -1 so DOS will recognize the structure as a device driver.

Device attributes

The second field is a bit field that's used to describe the device driver. Bit 15 of this field tells DOS whether this is a block or a character driver. A value of 0 represents a block driver and 1 represents a character driver. The interpretation of all other bits in this field depends on the setting of bit 15. The following is the entire structure of this bit field:

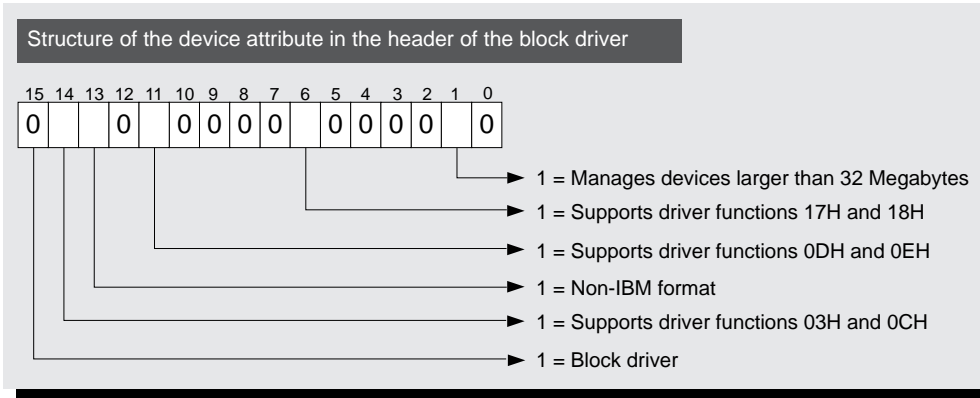
Device driver header		
Address	Contents	Type
00H	Offset address of next driver	1 word
02H	Segment address of next driver	1 word
04H	Device attribute	1 word
06H	Offset address of strategy routine	1 word
08H	Offset address of interrupt routine	1 word
0AH	Driver name (character driver) or number of devices (block driver)	8 bytes



In the case of a character driver, bits 0 through 3 identify the driver (see the previous illustration). They indicate whether the new driver, instead of the CON driver, should be used for input and output or whether it should replace the NUL driver or the clock driver. If none of these bits are set, then the new driver doesn't replace any of the standard drivers.

The other bits indicate whether the driver supports various driver functions. To understand this, you must also know that DOS recognizes 15 different driver functions, all of which aren't automatically available to a driver. The optional functions 03H, 0CH, 0DH, 0EH and 10H must be explicitly requested by setting the corresponding bits. Some of these functions are available beginning only with DOS Version 3.1 and some aren't available until Version 5.0. You must be using the corresponding DOS version for the attribute bits to be properly recognized. We'll provide descriptions of these DOS driver functions later in this chapter.

Remember the values of these bits are valid only if bit 15 contains the value 1b, which indicates a character driver. These bits will have different meanings if it's a block driver.



Besides the bits indicating which driver functions are supported, bit 1 is very important to block drivers. The feature indicated by this bit is supported in DOS Versions 4.0 and higher. If this bit is set, the block driver is capable of supporting devices and partitions larger than 32 megabytes, which was made possible with the enlarged cluster size available starting with DOS 4.0.

With the 32 megabyte limit exceeded, the driver can no longer represent all the addressable sectors as 16-bit integers. This would allow for only 65,536 sectors, which, with a standard sector size of 512 bytes, corresponds to exactly 32 megabytes. Device drivers with attribute bit 1 set expect sector numbers to be passed in another format so the sectors beyond the 32 megabyte limit can also be addressed. We'll provide more information about this when we discuss the driver functions.

Structure of the fields in the device driver header

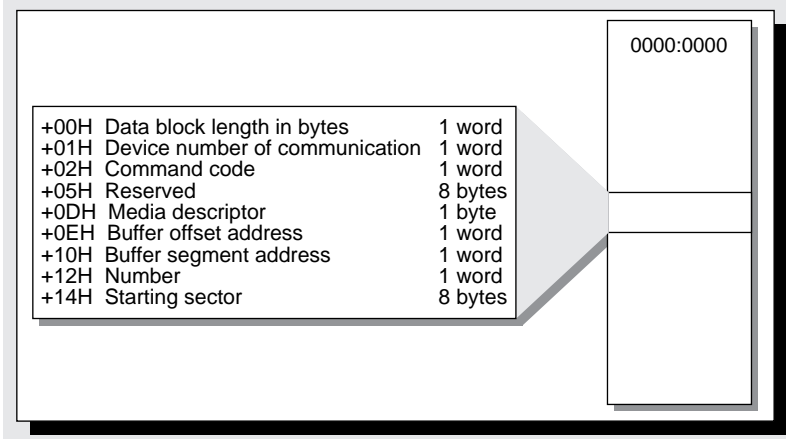
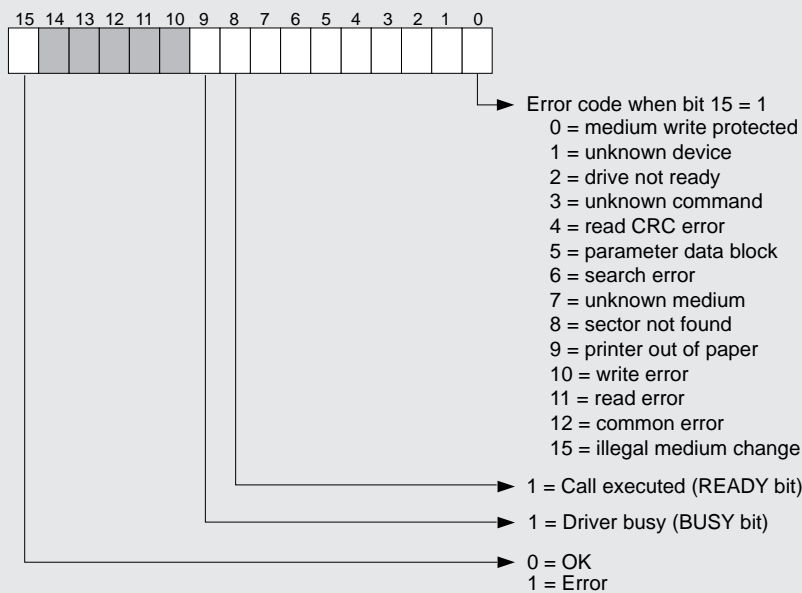
Two fields that contain the offset addresses of the so-called strategy and interrupt routines follow the attributes field. DOS uses these routines to communicate with the driver. Only the offset addresses of these routines are required because device drivers, like COM programs, are limited to one segment. So, the explicit segment address isn't needed.

In the case of a character driver, the last field of the driver header contains the name of the device driver. If the name is less than the eight characters allowed in this field, the rest of the field must be filled with empty spaces (ASCII code 32). If the driver is a block driver, then this field contains the number of logical devices the driver supports. The other 7 bytes of this field should then be filled with the value 0.

Strategy and interrupt routines

DOS calls the strategy routine to initialize the driver before any function of the driver is called. An address is passed to this routine in the ES:BX register pair. This address points to a data structure that contains information about the operation to be performed and the corresponding data. The strategy routine doesn't execute these operations itself; it simply stores the address of the data block and gives control back to DOS. Then the driver interrupt routine is called and the operation is actually executed. We'll learn more about this later.

This mechanism frees DOS from having to know the address of every driver function. DOS can simply use the strategy and interrupt routines as an interface to the various driver functions. Therefore, the data block whose address is passed to the strategy routine will always also contain the number of the driver function that must be called. The data block always consists of at least 13 bytes. More bytes can be added, depending on the needs of the function being called. The following figure shows a typical structure for this data block. The first 13 bytes will appear with all function calls.

Structure of the request header**Status field error codes****Device Driver Functions**

Starting with DOS Version 2.0, any installable device driver must support 13 functions, numbered from 00H to 0CH, even if their only action consists of setting the DONE flag in the status word. DOS Versions 3.0, 4.0 and 5.0 include additional functions that can be supported, but aren't required. Some of these functions concern one of the two driver types, while others apply to both driver types (e.g., initialization). Unused functions must at least set the DONE flag of the status word. Let's look at the various functions in detail according to their function numbers.

Request header

Every function described here receives its arguments from the request header (whose address is passed by DOS to the strategy routine) and stores its "results" in the request header. Therefore, the offset address to the arguments, relative to the beginning of the request header, is passed to the specified function. These arguments are later transferred to variables. Besides this offset address, a flag indicates whether this information consists of a byte, word or PTR. The PTR data type represents a pointer to a buffer and consists of two adjacent words. The first word is the offset address of the buffer. The second word is the segment address of the buffer.

Function 00H:	Driver Initialization
---------------	-----------------------

During the system boot procedure, DOS calls this function to initialize the device driver. This function can involve hardware initialization, setting various internal variables to their default values or the redirection of interrupts. Since the entire operating system hasn't been completely initialized at this point, the initialization routine can only call functions 01H through 0CH (character I/O) and 30H (DOS version number), 25H and 35H (get/set interrupt vector) of DOS interrupt 21H. These functions can be used to determine the DOS version number and to display a driver identification message on the screen. Even if the newly linked driver is a CON driver, the output to the display occurs through the old CON driver, because there are no new drivers linked into the system after the initialization routine is complete.

Initialization and the request header

The initialization routine can obtain two pieces of information from the request header. First, it provides the memory address, which contains the text that follows the equal sign on the line in the CONFIG.SYS file that loaded the driver into the system. A typical line in a CONFIG.SYS file can look like this:

```
DEVICE=ANSI . SYS
```

In this instance, the device name is ANSI.SYS, which assigns the standard ANSI escape sequences for screen control to the PC. The memory address passed to the initialization routine points to the character following the equal sign (in this case, the A of ANSI.SYS). This makes it possible to store additional information following the name of the device driver. This information is ignored by DOS, but can be read by other routines.

Address	Contents	Type
Input parameters		
Offset 2	Function number (0H)	byte
Offset 18	Address of character that follows the equal sign after the DEVICE	ptr
Offset 22	Device number of the first device supported by the driver (0=A,	byte
Returned parameters		
Offset 3	Status word	word
Offset 13	Number of devices supported (block devices only)	byte
Offset 14	Address of first available memory location following the driver	ptr
Offset 18	Address of array containing BPB addresses (block devices only)	ptr

Logical device designation

The second item is only available under DOS Version 3.0 and higher and only if the driver is a b Block device driver. This is the letter designation of the first logical device of the driver. The value 0 represents A, 1 represents B, 2 represents C, etc.

The initialization routine must return four parameters to the calling DOS function. The first parameter is the status of the function (i.e., the indication of whether the function has executed properly). For a block device driver, the number of logical devices supported must also be passed. This information could also be obtained from the device driver's header, but is ignored by DOS.

The next parameter the device driver must pass to DOS is the highest memory address that it occupies or uses. This lets DOS know where the next device driver can be installed. Remember, starting with DOS Version 5.0, drivers no longer have "unlimited" memory available to them. This DOS version allows you to load drivers into Upper Memory Blocks. Generally, there is less memory available there than in the TPA below the 640K limit. In Version 5.0, the "upper limit" address of the driver is loaded in the field where DOS normally would find the end address of the driver. The upper limit is the address in memory beyond which the driver may no longer allocate memory for itself. If this isn't sufficient for the driver, then it shouldn't even attempt to be installed.

BPB

If the driver is a block device driver, the last argument passed must be the address of an array that contains an entry for every logical device. This array contains the addresses of BIOS parameter blocks (BPBs). The address is passed as two words; the first word contains the offset and the second word contains the segment address of the array. The first two words within this table are the address for the first logical device supported. The next two words indicate the address for the second logical device, etc. The BPB is a data block containing information that describes a logical device. If all or some of the logical devices have the same format, all entries in the BPB address table can point to a single BPB.

Notice the last part of the BIOS parameter block (called the expanded BPB). It's only required for block drivers supporting partitions larger than 32 megabytes. This is only possible with DOS 4.0 or higher. To indicate that you want to manage partitions larger than 32 megabytes, set bit 1 in the driver attribute bit field. The total number of sectors and the number of reserved sectors in the normal BPB must be set to 0 and the corresponding fields in the expanded BPB must be loaded. These are 32 bit fields that can handle values greater than 65,535 (which is the limit for these fields in the normal BPB).

BIOS Parameter Block design		
+00h	Bytes per sector	1 word
+02h	Sectors per cluster	1 byte
+03h	Reserved sectors (including boot sectors)	1 word
+05h	Number of FATs	1 byte
+06h	Maximum number of entries in root directory	1 word
+08h	Total number of sectors	1 word
+0Ah	Media descriptor	1 byte
+0Bh	Number of sectors per FAT	1 word

Media descriptor byte	
Code	Medium
F0h	3.5-inch diskette, double-sided, 80 tracks, 18 sectors per track 3.5-inch diskette, double-sided, 80 tracks, 36 sectors per track
F8h	Hard drive
F9h	3.5-inch diskette, double-sided, 80 tracks, 9 sectors per track 5.25-inch diskette, double-sided, 80 tracks, 15 sectors per track
FAh	3.5-inch diskette, single-sided, 80 tracks, 8 sectors per track 5.25-inch diskette, single-sided, 80 tracks, 8 sectors per track
FBh	3.5-inch diskette, double-sided, 80 tracks, 8 sectors per track 5.25-inch diskette, double-sided, 80 tracks, 8 sectors per track
FCh	5.25-inch diskette, single-sided, 40 tracks, 9 sectors per track
FDh	5.25-inch diskette, single-sided, 40 tracks, 9 sectors per track
FEh	5.25-inch diskette, single-sided, 40 tracks, 8 sectors per track
FFh	5.25-inch diskette, single-sided, 40 tracks, 8 sectors per track

Starting with DOS Version 4.0, the function can also return an error flag to offset address 17H of the data block. If DOS finds a value other than 0 at this location after the function call, the message "CONFIG.SYS error in line xx" is displayed.

Function 01H:	Media Check
---------------	-------------

This function is used only with a block device driver. A character device driver should simply set the DONE flag of the status word and exit. This function is used by DOS to determine whether the media (diskette) has changed. It's frequently used when

examining a disk directory. If the disk medium wasn't changed since the last access, DOS still has this information in memory; otherwise DOS must reread the information from the media that delays the execution of the current task.

In some instances, as with floppy diskettes, the answer to the question is fairly complicated. Therefore, DOS permits function 1 to answer not only with "yes" and "no", but also with "don't know." The answer always affects further DOS activity. If the media is unchanged, access to the media can occur immediately. If the media was changed, however, DOS closes all internal buffers related to the current logical device. This causes the loss of all data that should have been transmitted to the media. Then it calls function 2 of the current device driver and loads the FAT and the root directory.

If the media check function answers with "don't know," the additional steps taken by DOS depend on the status of the internal buffers related to the current logical device. If these internal buffers are empty, DOS assumes the media was changed and acts as if function 1 answered "yes." If the buffers contain data that should have been transmitted to the media, DOS assumes the media is intact and writes the data. If the media was indeed changed, the data written to a changed media may damage the new diskette's file structure.

Since subsequent processing depends on the response from the media check function, the driver should handle the response carefully. Before enabling the mechanism used by the function to respond, the function examines the parameters passed to it. If the driver supports several logical devices, the first parameter is the number of devices. Next is a media descriptor code. This code contains information about the type of media last used in the current logical device. Only devices that can handle several different formats can use this task (e.g., AT disk drives that can use both 360K and 1.2 megabyte diskette formats).

If the media check function determines the medium in a device is non-removable (e.g., a fixed disk), it can always respond "not changed". If, however, the device media can be changed (e.g., a disk), the correct response can only be determined with complex procedures. If these procedures aren't used, the response should be "don't know".

Address	Contents	Type
Input parameters:		
Offset 1	Device number	byte
Offset 2	Function number (1)	byte
Offset 13	Media descriptor byte	byte
Returned parameters:		
Offset 3	Status word	word
Offset 14	Was media changed? FFH = yes, 00H = don't know, 01H = no	byte
Offset 15	Address of buffer containing previous volume name (only if device indicates a media change)	ptr

The following are the three procedures that provide fairly accurate results. Since a device with changeable media has an opening and closing mechanism, the function should check to determine whether the media was removed. However, it cannot determine whether the removed media is identical to the newly inserted medium. If the media has a name, the function should read this name to determine if the media was changed. This procedure only makes sense if every media has a unique name.

The disk drive procedure used by DOS is based on the fact that changing medium is time-consuming. DOS assumes that a user needs about two seconds to remove a diskette from a drive and insert a new diskette in the same drive. If two consecutive diskette accesses occur less than two seconds apart, DOS assumes the diskette wasn't changed. A byte in the data block is used to indicate changes. The value -1 (FFH) means "changed", 0 means "don't know" and 1 means "not changed". If the media was changed, the device driver indicates this (bit 11 in the device attribute = 1); the address of a buffer must be passed to DOS Version 3 and higher, which contains the volume name of the previous media. This name must be stored there as an ASCII string and terminated with an end character (ASCII code 0).

Function 02H:	Build BIOS Parameter Block (BPB)
---------------	----------------------------------

This function is used only by block device drivers. A character device driver should just set the DONE flag of the status word and exit. DOS calls this function when the media check function determines the media was changed. This function returns

a pointer to a new BPB for the media. As you can see by the layout of the calling parameters, the device number media descriptor and a pointer to a buffer are passed to this function by DOS. If the device is a standard format (bit 13 of the device attribute =0), then the buffer contains the first sector of the FAT. Starting with DOS Version 3.0, this function also reads and writes volume label names, because a call to function 01H must occur any time a device experiences a change of medium (bit 11 of the device attribute = 1).

Address	Contents	Type
Input parameters		
Offset 1	Device number	byte
Offset 2	Function number (2)	byte
Offset 13	Media descriptor byte	byte
Offset 14	Address of a buffer containing the FAT (see above)	ptr
Returned parameters		
Offset 3	Status word	word
Offset 18	Address of the BPB of addressed device	ptr

Function 03H: I/O Control Read

This function allows direct communication between a device driver and an application. This allows device drivers to implement additional logic not accessible from normal driver functions. It can only be called through function 44H of interrupt 21H if the IOCTL bit (bit 14), in the device attribute word in the device driver header, is set. Different parameters are passed to the function from a FAR pointer, depending on whether the driver is a character or a block device driver.

A character device driver is passed the number of characters to be transferred and the address of a buffer for the transfer of the data. A block device driver is passed the device number, the media descriptor byte, the address of the buffer to be used for the data transfer, the pointer to the first sector to be read and the number of sectors to be read.

Address	Contents	Type
Input parameters:		
Offset 1	Device number (block devices only)	byte
Offset 2	Function number (3)	byte
Offset 13	Media descriptor byte (block devices only)	byte
Offset 14	Address of buffer into which data should be transmitted	ptr
Offset 18	Number of sectors to be read (block device) or number of characters to be read (character device)	word
Offset 20	First sector to be read (block devices only)	word
Returned parameters:		
Offset 3	Status word	word
Offset 18	Number of sectors read (block device) or number of characters read (character device)	word

Function 04H: Read

This function reads data from the device to a buffer specified in the calling parameter. If an error occurs while reading the data, the error status must be set. Also, the function must report the number of sectors or bytes that were successfully read. Simply reporting an error isn't sufficient.

Block drivers have different methods of passing the sector number. The methods used depends on whether the driver is a 16-bit or a 32-bit driver. For 16-bit drivers, the sector number is stored in the data block as a word starting at offset address 14H. For 32-bit drivers, it is a dword starting at offset address 1AH. A 32-bit driver refers to a driver that can handle devices and

partitions larger than 32 megabytes. This is indicated by setting bit 1 in the driver attribute bit field. In addition to this, the sector number at offset address 14H is set to FFFFH for 32-bit drivers to indicate the use of the 32-bit sector number at 1AH.

Address	Contents	Type
Input parameters:		
Offset 1	Device number (block device only)	1 byte
Offset 2	Function number (4)	1 byte
Offset 13	Media descriptor byte (block device only)	1 byte
Offset 14	Address of buffer to which data should be read	1 ptr
Offset 18	Number of sectors to be read (block device) or Number of characters to be read (character device)	1 word
Offset 20	First sector to be read (block device only)	1 word
Returned parameters:		
Offset 3 (word)	Status word	1 word
Offset 18 (word)	Number of sectors read (block device) or Number of characters read (character device)	1 word
Offset 22 (ptr)	Pointer to volume ID on return of error 0FH (Version 3.0 and higher)	1 ptr

Function 05H:	Non-destructive Read
---------------	----------------------

This function is used by a character device driver to test for unread characters in the input buffer. A block device should set the DONE flag of the status word and exit.

DOS tests for additional characters using this function. If more characters exist, the busy bit must be cleared (set to 0) and the next character passed to DOS. The character that is passed remains in the buffer so that a subsequent call to a read function will return this same character. If additional characters don't exist, the busy bit must be set (set to 1).

Address	Contents	Type
Input parameters:		
Offset 2	Function number (5)	1 byte
Returned parameters:		
Offset 3	Status word	1 word
Offset 13	The character read	1 byte

Function 06H:	Input Status
---------------	--------------

This function is used to determine whether a character is waiting to be read from the input buffer of a character device. A block device driver should set the DONE flag of the status word and exit.

If a character is waiting to be read from the input buffer, the busy bit is cleared (set to 0). If a character isn't in the input buffer, the busy bit is set (set to 1). When a character is waiting to be read, the Input Status function (06H) resets the status word busy bit to 0 and returns the character to DOS. The character isn't removed from the buffer and is therefore nondestructive. This function is equivalent to reading one character ahead.

Address	Contents	Type
Input parameters		
Offset 2	Function number (6)	1 byte
Returned parameters		
Offset 3	Status word: 0 = Characters already in buffer 1 = Read request to physical device	1 word

Function 07H:	Flush Input Buffers
---------------	---------------------

This function clears the internal input buffers of a character device driver. Any characters read but not yet passed to DOS are lost when this function is used. A block device driver should set the DONE flag of the status word and exit.

Address	Contents	Type
Input parameters:		
Offset 2	Function number (7)	1 byte
Returned parameters:		
Offset 3	Status word	1 word

Function 08H: Write

This function transfers characters from a buffer to the current device. If an error occurs during transmission, the status word is used to indicate this error. Both block and character devices use this function. The parameters used for this function depend on whether the driver is for a character or block device. Both pass a buffer address, from which a certain number of characters should be transferred. A character device driver is passed the number of bytes to be transferred in addition to this information.

A block driver is passed the number of sectors to transfer (not the number of characters), the number of the device to be addressed, its media descriptor and the address of the first sector on the medium. If an error occurs writing the data, the error status must be set. Also, the function must report the number of sectors or bytes written successfully. Simply reporting an error isn't sufficient.

Address	Contents	Type
Input parameters		
Offset 1	Device number (block drivers only)	1 byte
Offset 2	Function number (8)	1 byte
Offset 13	Media descriptor of device addressed (block device only)	1 byte
Offset 14	Address of the buffer containing data	1 ptr
Offset 18	Number of sectors to be written (block device) or Number of characters to be written (character device)	1 word
Offset 20	First sector to be written (block device only)	1 word
Returned parameters		
Offset 3	Status word	1 word
Offset 18	Number of sectors written (block device) or Number of characters written (character device)	1 word
Offset 22	Pointer to volume ID on return of error 0FH (Version 3.0 up)	1 ptr

Function 09H: Write with Verify

This function is similar to function 08H, except the characters written are reread and verified. Some devices, especially character devices, such as a monitor or a printer, don't require verification because either no errors occur during transmission (monitor) or the data cannot be verified (printer).

Address	Contents	Type
Input parameters		
Offset 1	Device number (block drivers only)	1 byte
Offset 2	Function number (09H)	1 byte
Offset 13	Media descriptor of device addressed (block device only)	1 byte
Offset 14	Address of the buffer containing data	1 ptr
Offset 18	Number of sectors to be written (block device) or Number of characters to be written (character device)	1 word
Offset 20	First sector to be written (block device only)	1 word
Returned parameters		
Offset 3	Status word	1 word
Offset 18	Number of sectors written (block device) Number of characters written (character device)	1 word
Offset 22	Pointer to volume ID on return of error 0FH (Version 3.0 up)	1 ptr

Function 0AH: Output Status

This function indicates whether the last write operation to a character device is completed. A block device should set the DONE flag in the status word and exit. If the last write operation is complete, then the busy bit of the status word is cleared; otherwise the busy bit is set to 1.

Address	Contents	Type
Input parameters		
Offset 2	Function number (0AH)	1 byte
Returned parameters		
Offset 3	Status word: BUSY bit = 1 if the last character output hasn't been completed	1 word

Function 0BH: Flush Output Buffers

This function completely clears the output buffer even if it contains characters waiting for output. A block device should set the DONE flag on the status word and exit.

Address	Contents	Type
Input parameters		
Offset 2	Function number (0BH)	1 byte
Returned parameters		
Offset 3	Status word	1 word

Function 0CH: I/O Control Write

This function passes control information from the application program to the character or block device driver. It can only be called through function 44H of interrupt 21H, if the IOCTL; bit in the device attribute word in the device driver header is set. Different parameters are passed to the function, depending on whether the driver is a character or a block device driver.

Address	Contents	Type
Input parameters		
Offset 1	Device number (block device only)	1 byte
Offset 2	Function number (0CH)	1 byte
Offset 13	Media descriptor of addressed device (block device only)	1 byte
Offset 14	Address of buffer from which data should be read	1 ptr
Offset 18	Number of sectors to be written (block device) or Number of characters to be written (character device)	1 word
Offset 20	First sector to be written (block device only)	1 word
Returned parameters		
Offset 3	Status word	1 word
Offset 18	Number of sectors written (block device) or Number of characters written (character device)	1 word

A character device driver is passed the number of characters to be written and the address of the buffer from which these characters are transferred. A block device driver is passed the device number (in case the driver services logical devices), the media descriptor byte, the address of the buffer from which the data is to be written, the number of the first sector to be written and the number of sectors to be written. A character device driver returns the number of bytes written. A block device driver returns the number of sectors written.

The following four functions are supported by DOS Version 3.0 and higher

Function 0DH:	Open
---------------	------

This function can be used only if the OCR (Open/Close/RM) bit in the device attribute word in the device driver header is set. Its task differs, depending on whether it's a character or block driver.

A block driver uses this function every time a file is opened. This function determines how many open files exist on this device. Use this command carefully, since programs that access FCB function calls usually don't close open files. This problem can be avoided by assuming, during every media change, that no files remain open. For devices with non-changeable media (e.g., a hard drive), even this procedure may not help.

Within a character driver, this function can send an initialization string to the device before transmitting the data. This is helpful when used to communicate with the printer. The initialization string shouldn't be included in the driver, but can be called, for example, with the IOCTL function of interrupt 21H, which calls function 0CH of a driver to transmit it from an application program to the driver. The function can also be useful because it can prevent two processes (in a network or in multiprocessing) from both accessing the same device.

This function isn't called for the devices CON, PRN and AUX because these functions are always open.

Address	Contents	Type
Input parameters		
Offset 1	Device number (block device only)	1 byte
Offset 2	Function number (0DH)	1 byte
Returned parameters		
Offset 3	Status word	1 word

Function 0EH:	Device Close
---------------	--------------

This function is the opposite of function 0DH. This function can only be addressed if the OCR bit in the device attribute word of the device driver header is set. Its task differs, depending on whether it's a character or block driver.

A block driver calls it after closing a file. This can be used to decrement a count of open files. Once all files on a device are closed, the driver should flush the buffers on removable media devices, because it's likely the user is about to remove the media. A character driver can use this function to send some closing control information to a device after completing output. For a printer, this could be a formfeed. As in function 0DH, the string could be transmitted from an application using the IOCTL function.

Address	Contents	Type
Input parameters		
Offset 1	Device number (block device only)	1 byte
Offset 2	Function number (0EH)	1 byte
Returned parameters		
Offset 3	Status word	1 word

Function 0FH:	Removable Media
---------------	-----------------

This function indicates whether the media in a block device can be changed. This function is used only if the OCR bit in the device attribute word of the device driver is set. A character device driver should set the DONE flag in the status word and exit. If the media can be removed, the busy bit is cleared; otherwise it is set to 1.

Address	Contents	Type
Input parameters		
Offset 1	Device number (block device only)	1 byte
Offset 2	Function number (0FH)	1 byte
Returned parameters		
Offset 3	Status word: If the media can be removed, the busy bit must contain the value 0	1 word

Function 10H: Output Until Busy

Transfers data from a buffer to an output device until the device is busy (i.e., can no longer accept more characters). Since this function is supported by character devices, a block device driver should set the DONE flag on the status word and exit.

This function works particularly well with print spoolers, through which files can be sent to a printer as a background activity while a program executes in the foreground. It's possible that not all the characters in the transfer request will be sent to a device during this function call. This usually isn't an error; it could be the result of the device becoming busy. The function is passed the number of characters to be transmitted as well as the buffer address. If, during transmission, the output device indicates that it can no longer accept additional characters, it indicates the number of characters successfully transferred and returns control to the device driver.

Address	Contents	Type
Input parameters		
Offset 2	Function number (10H)	1 byte
Offset 14	Address of buffer from which data should be read	1 ptr
Offset 18	Number of characters to be read	1 word
Returned parameters		
Offset 3	Status word	1 word
Offset 18	Number of characters written	1 word

The following functions are supported by DOS Version 4.0 and higher.

Function 17H: Get Logical Device

This optional function can only be used with block drivers. If a block driver contains this function, bit 6 of the device attribute bit field must be set so that DOS will know it's available. This function is only useful in conjunction with the DOS device driver DRIVER.SYS, which enables a diskette drive to use two different formats. This function is used to tell the caller which of the two formats is currently in use.

The device code isn't required as a parameter for this function. By definition, only one driver with a switchable disk drive is allowed within a system.

Address	Contents	Type
Input parameters		
Offset 0	Number of bytes requested	1 byte
Offset 1	Device number (block devices only)	1 byte
Offset 2	Function number (17H)	1 byte
Returned parameters		
Offset 3	Status word	1 word

Function 18H: Set Logical Device

This function is the opposite of function 17H. DOS uses it to tell the device driver the diskette drive it manages has another drive letter that can be addressed with a different diskette format.

Address	Contents	Type
Input parameters		
Offset 0	Number of bytes requested	1 byte
Offset 1	Device number (block devices)	1 byte
Offset 2	Function number (18H)	1 byte
Returned parameters		
Offset 3	Status word	1 word

Clock Drivers

The clock driver is a character device driver whose only function is to pass the date and time from DOS to an application. The clock driver can also have a different name (such as \$CLOCK). This is possible because DOS identifies it by the fact that bit 2 in the device attribute word of the device driver header is set to 1, instead of by name. Bit 15 must also be set since the clock driver is a character device driver. Functions 2AH to 2DH of DOS interrupt 21H read the date and time and call the driver. A clock driver must support only functions 4, 8 and 0 (initialization). During the call of function 4 (reading), the date and time pass from the driver to DOS. DOS can set a new date and time with function 8. Both functions have the time and date passed in a buffer that is 6 bytes long. The date format is unusual. Instead of passing the month, day and year separately, DOS passes the number of days elapsed since January 1, 1980 as a 16-bit number. A fairly complex formula converts this number into normal date format, taking leap years into account. The clock driver normally uses function 0 and 1 of the BIOS interrupt 1AH to read and set the time.

Clocks on AT models

AT and AT-compatible computers have a battery powered realtime clock. Functions 0 and 1 of interrupt 1AH use a software controlled time counter instead of the battery powered realtime clock. When the computer is rebooted, the date and time previously set with driver function 8 is cleared. You can use the clock driver to access the realtime clock using functions 2 and 5 of interrupt 1AH instead of function 0 and 1.

Device Driver Calls From DOS

Now that you're familiar with the functions of the different device drivers, you can develop your own personal device driver. The following steps are performed before and after calling a device driver function. A chain of events begins when a DOS function, which handles input and output, is called using interrupt 21H. Calling one of these functions can, in turn, call a series of other functions and corresponding read and write operations.

An example of this is when the Open function 3DH is called to open a file in a subdirectory. Before it can be opened, DOS must find the file. This may require a search of a set of directories instead of simply reading in the FAT. During each access of interrupt 21H, DOS determines which of the available device drivers should be used to read or write characters. When this happens, DOS sets aside an area in memory to store the information required by the device driver.

For files, DOS must convert the number of records to be processed into logical sector numbers. DOS then calls the strategy routine of the device driver, to which it passes the address of the newly created data block (request header). Then the interrupt routine of the driver, which stores all registers, is called. It isolates the function code of the requested function from the data block and starts to process the function. If the addressed driver is a character device driver, the function only has to send the characters to the hardware or request the characters to be read.

For a block device (e.g., a mass storage device such as a floppy or hard disk), the logical sector number must be converted into a physical address before a read or write access. The logical sector number is divided into a head, track and physical sector number. After the read or write operation ends, the driver function must place a result code in the status field of the request header to be returned to the calling DOS function. Next, the contents of all registers are restored and control is returned to the calling DOS function, which, depending on the result of the driver function, sets or resets the carry flag and places any error code into the AX register. The interrupt function then returns control to the routine that called interrupt 21H.

Direct Device Driver Access: IOCTL

Now we'll discuss IOCTL in detail, because it offers an alternate method of communicating with the device driver. You can only use these functions if the IOCTL bit (bit 14) of the device attribute is set. The IOCTL function itself is one of many functions addressable from DOS interrupt 21H. Its function number is 44H. Three groups of subfunctions are accessible:

- Device configuration
- Data transmission
- Driver status

The number of the desired subfunction is passed to the IOCTL function in the AL register. After the function call, the carry flag indicates whether the function executed correctly. A set carry flag indicates that an error occurred and the error code is located in the AX register.

Character device drivers status

The number of the desired subfunction is passed to the IOCTL function in the AL register. After the function call, the carry flag indicates if the function executed correctly. A set carry flag indicates that an error occurred and the error code is located in the AX register. Subfunctions 06H and 07H can determine the status of a character device driver. Subfunction 6 can determine if the device is able to receive data. Subfunction 7 can determine if the device can send data. The handle of this device is passed in the BX register. If the device is ready, both functions 06H and 07H return the value FFH in the AL register.

Subfunction 02H reads control data from the character device driver. The handle is passed in the BX register and the number of bytes to be read is passed in the CX register. Also, the DS:DX register pair contain the address of the buffer into which the data will be read. If the carry flag is clear, then the function was successful and the AX register contains the number of characters read. If the carry flag is set, then there was an error and the AX register contains the error code.

Subfunction 03H writes control information from a buffer to the character device driver. Again, the handle is passed in the BX register, the number of bytes to be written in the CX register and the address of the buffer in the DS:DX register pair. The return codes are the same as for subfunction 02H. These two subfunctions are used to pass information between the application program and the device driver.

Block device driver status

Subfunctions 04H and 05H have the same task as subfunctions 2 and 3. However, they are used for block devices instead of character devices. Instead of passing the handle in register BX, you pass the drive code (0=A, 1=B, etc.) in the BL register. Subfunction 0 is used to obtain device information for a specified handle. The subfunction number is passed in the AL register and the handle in the BX register. The function returns the device information word in the DX register.

Passing date and time to a clock driver		
Address	Contents	Type
+ 00H	Number of days since January 1, 1980	1 word
+ 02H	Minutes	1 byte
+ 03H	Hour	1 byte
+ 04H	Hundredths of seconds	1 byte
+ 05H	Seconds	1 byte
Length: 6 bytes		

For block devices:	
Bit(s)	Function
8-15	Reserved
7	0 if a block device
6	0 if file has been written 1 if file has not been written
0-5	Drive code (0=A, B=1, etc.)

For character devices:	
Bit(s)	Function
15	Reserved
14	1 if device supports IOCTL subfunctions 0 if device does not support IOCTL subfunctions
8-13	reserved
7	1 if a character device
6	0 if end of file for input device
5	0 if cooked mode 1 if raw mode
4	Reserved
3	1 if clock device
2	1 if NUL device
1	1 if standard output device
0	1 if standard input device

Cooked and raw modes

Subfunction 01H is used to set device information for a specified handle. This subfunction is often used to set the standard input device from cooked mode to raw mode or back.

Two final interrupts are sometimes used by block device drivers. These two interrupts, 25H and 26H, are used to read from and write to the disk drive. You can use these interrupts, for example, to process disks that were formatted using a "foreign" operating system. The device number is passed in the AL register, the number of sectors to be transferred is passed in the CX register, the starting sector number to be transferred is passed in the DX register and the buffer is passed in the DS:BX register pair. The carry flag is clear if no errors occurred. If the carry flag is set, then the error code is returned in the AX register.

The operation of these two interrupts had to be changed with DOS 4.0 because the 16-bit sector number in the DX register can only be used to address 65,535 sectors, which does not allow access beyond the 32 megabyte limit. Starting with DOS 4.0, these interrupts are passed a pointer to a data block in the DS:BX register pair (see the illustration below). This allows access to larger volumes and partitions. Also, the value -1 (FFFFH) must be passed in the CX register. This informs DOS the new parameters are being used and that it shouldn't search for parameters according to the old scheme. The old scheme can still be used for volumes larger than 32 megabytes as long as only sectors below the 32 megabyte limit are addressed.

Tips On Developing Device Drivers

When you're developing a device driver, problems occur when you test the new driver. First, a device driver must load into a memory location assigned to it by DOS, at an address unknown to the programmer. Also, a newly developed CON driver can't be tested using the DEBUG program, because DEBUG uses this driver for character input and output. After you write the actual driver, you should write a short test program that calls the individual functions in the same way as DOS, but without having the driver installed as part of DOS. By doing this, everything executes under user control and the entire process can be corrected with a debugger. Remember, a new device driver (especially a block device driver) should be linked into the system only after it's been tested completely and proven to be error-free.

Sample Device Drivers

This section describes a sample device driver for each of the three types of device drivers. This sample demonstrates the information you've read so far. We'll talk about an alternative console driver, a 160K RAM disk and a driver for a battery operated AT clock. All these drivers must follow the same rules set for a COM program. They must comprise a single segment containing program code and driver data. No direct segment references can exist. Unlike a COM program, the device driver must begin at offset 0H (not 100H), because a PSP cannot exist in memory.

The first program is a character driver that corresponds exactly to the format of a normal console driver. The second program is a block device driver, which creates a 160K RAM disk. The final program is a DOS clock driver to support an AT computer realtime clock. The header of this driver describes a character device driver that handles both the standard input device (keyboard) and the standard output device (monitor). After linking it into the system, setting the two bits in the device attribute calls this driver on all function calls previously handled by the CON driver. Like any other driver, this driver has a strategy routine and an interrupt routine. The strategy routine stores the address of the data block in the variable DB_PTR.

The interrupt routine saves the contents of all registers that will be changed by it on the stack and obtains the routine number to be called from the data block. It then checks whether CONDRV supports this function. If it doesn't, it jumps directly to the end of the interrupt routine and sets the proper error code in the status field of the request header that was passed to the routine. Then it restores the registers that were saved on the stack and returns control to the calling DOS function.

For any of the functions that are supported by the device driver, the offset address of a routine to handle a particular function is determined from the table labeled FCT_TAB. Notice the routines named DUMMY and NO_SUP appear several times. DUMMY is for all functions that apply only to block device drives and, therefore, aren't used in this driver. The DUMMY routine clears the AX register and sets the BUSY bit in the status word. The NO_SUP routine handles any functions that cannot be used since the drive attribute for CONDRV doesn't support these functions.

The `STORE_C` routine can be accessed from the lower level routines in this driver. Its purpose is to store a character in the internal keyboard buffer of the driver. The driver really shouldn't have this buffer available since BIOS (whose functions are used by the driver to read characters from the keyboard) also has this type of buffer. The problem is the BIOS always returns two characters when pressing a key with extended codes (cursor keys, function keys, etc.). If the higher level functions of DOS only ask for one character at a time from `CONDRV`, the second character shouldn't be lost. Instead, it should be stored in a buffer and delivered to DOS by the read function on the next call. This is `STORE_C`'s task.

Reading characters

The next routine is the `READ` function. It obtains the number of characters to be read from the request header passed by DOS. The routine is terminated immediately if 0. If it isn't 0, then a loop starts and executes once for every character read. It first tests for characters still stored in the internal keyboard buffer. If characters are found, a character is passed to the buffer of the calling function. If more characters exist in the keyboard buffer, function 0 of the BIOS keyboard interrupt 16H inputs a character from the keyboard. It's also passed to the internal keyboard buffer. If it's an extended keycode, it's divided into two characters. The next step removes a character from the internal keyboard buffer and passes the character to the buffer of the calling function. The process repeats until all characters requested have been passed to DOS. Then the routine ends.

The higher level DOS functions also call the function named `READ_P`. It tests whether a character was entered from the keyboard. If not, it sets the `BUSY` bit in the status field of the request header passed by DOS and returns to the calling function. If a character was entered without having been read, the driver reads this character and passes it to the calling DOS function in the request header and resets the busy bit. The character remains in the keyboard buffer and on a subsequent call the read function, it's again passed to DOS. To test the availability of a character, the `READ_P` function uses function 1 of the BIOS keyboard interrupt 16H.

The function `DEL_IN_B` is also called by the higher level DOS functions. `DEL_IN_B` deletes the contents of the keyboard buffer. It removes characters from the buffer using function 0 of the BIOS keyboard interrupt until function 1 indicates that no more characters are available. This ends the function and it returns to the calling function after the `BUSY` bit is reset.

Writing characters

`WRITE` takes the number of characters from a buffer passed by DOS and displays the characters on the screen. This routine uses function 0EH of the BIOS video interrupt. Once all characters have been displayed, it sets the `BUSY` bit in the status field and ends the function. This function also executes when the higher level DOS functions call the `Write` and `Verify` functions.

Initialization

The last function, which is the initialization routine, is called first by DOS. Since `CONDRV` doesn't initialize variables and hardware, the routine simply enters the driver's ending address into the passed request header. The routine returns its own starting address since it will never be called again and is the end of the chain of drivers.

In its current form, the driver isn't very useful because it uses only those functions already available to the `CON` driver of DOS. It would be more practical if an enhanced driver, such as `ANSI.SYS`, was developed. An enhanced driver provides more control over the screen design. For example, it's possible that such a driver would have complete windowing capability, which could be accessed from any program, in any programming language.

The `RAMDISK.ASM` block device driver which you'll find on the companion CD-ROM creates a 160K RAM disk. This driver is similar to the `CONDRV` driver. The biggest difference between the two lies in the functions that each supports. First, this routine finds the DOS version number using function 30H. If the version number is equal to or greater than 3, the request header passed by DOS contains the device designation of the RAM disk. The system reads the designation, changes it to a

NOTE

When working with a hard drive, prepare a floppy system diskette before test booting the system from the hard drive with the new driver installed for the first time. If a small bug should exist in the new driver and the initialization routine hangs up, the booting process will not end and DOS will be out of control. In such a case, the only remedy is to reset the system and boot with a DOS diskette in the floppy disk drive. Once DOS loads, you can then access the hard drive and remove the new driver.

character and places the character into the installation message. DOS function 09H- is used to display this message on the screen.

Next, the program computes the ending address of the RAM disk. Since the actual data area of the RAM disk starts immediately after the last routine of this driver, 160K is added to the program's ending address. Also, the address of a variable (BPB_PTR) containing the address of the BIOS parameter block is passed to DOS. This variable describes the RAM disk's format. In this case, it tells DOS the RAM disk uses 512 bytes per sector. Each cluster consists of one sector and only one reserved sector (the boot sector) exists. In addition, only one FAT exists. Additional information indicates that a maximum of 64 entries can be made in the root directory and the RAM disk has 320 sectors available (160K of memory). The FAT occupies a single sector and the media descriptor byte FEH designates a diskette with one side and 40 tracks of 8 sectors each.

These parameters are then placed into the request header of DOS and the segment address of the data area of the RAM disk is calculated. (This information is needed by the driver itself instead of by DOS.)

The INIT routine

The RAM disk must now be formatted to create a boot sector, FAT and a root directory. Since these data structures are in the first sectors of the RAM disk, a normal INIT routine (which releases its memory to DOS), would overwrite itself with these data structures and would crash the system. This is why the initialization routine isn't at the end of the last routine of the driver, which would place it at the beginning of the RAM disk's data area.

The boot sector occupies the entire first sector of the RAM disk. However, only the first 15 words are copied into it because this is all DOS needs. The name "boot sector" is actually a misnomer here, because it's impossible to boot a system from a RAM disk. The second sector of the RAM disk contains the FAT. The first two entries are the media descriptor byte and 0 in the subsequent entries. These zeros indicate unoccupied clusters (an empty RAM disk). The last data structure is the root directory. It contains only the volume name.

Remaining routines

This concludes the work of the initialization routine and returns the system to the calling function. The remaining driver routines are explained in the order in which they appear. The DUMMY routine performs the same task as the routine of the same name in the CONDRV driver. The MED_TEST routine is found only in block device drivers. This routine informs DOS whether the medium was changed.

The next routine, GET_BPB, simply passes the addresses of the variables, which contain the address of the BPB of the RAM disk, to DOS, as the initialization routine had already done. NO_REM allows DOS to sense whether the medium (the RAM disk) can be changed. You cannot change a RAM disk, so the program sets the BUSY bit in the status field.

The two most important functions of the driver perform read and write operations. As in CONDRV, the program calls Write and Verify instead of the normal Write function, since a data error cannot occur during RAM access. The routine itself does very little; it loads the value 0 into the BP register and jumps to the MOVE routine. The READ routine performs in a similar way except that it loads a 1 into the BP register.

MOVE itself is an elementary routine for moving data. The BP register signals whether data is to move from the RAM disk to DOS or in the opposite direction. The routine receives all other data (the DOS buffer's address, the number of the sectors to be transferred and the first sector to be transferred) from the data block passed by DOS. See the comments in the MOVE routine for details of the procedure.

Changes

Obviously, this RAM disk can be enhanced. If you have enough unused memory, you can extend the size of the RAM disk to 360K. AT owners could make the RAM disk resident beyond the 1 megabyte boundary. In this case, the data transfer between DOS and the RAM disk would use function 87H of interrupt 15H.

You'll find the following program(s) on the companion CD-ROM



CONDRV.ASM (Assembler listing)

The clock driver

This final sample driver directly accesses the battery powered clock of an AT computer. It's useful because when the DOS commands DATE and TIME are used, the date and time are passed directly to the battery powered realtime clock. Reading the date and time reads the information directly from the memory locations of the realtime clock.

The basic structure of this driver differs from the other drivers because it calls the individual functions directly, instead of through a table of their addresses. Since it only supports functions 00H, 04H and 08H, this driver can test the function numbers directly passed by DOS. If any other function occurs, it signals an error. Besides the INIT routine, which sets only the ending address of the driver like CONDRV, the driver also contains the Read Time and Date and Write Time and Date functions.

You'll find the following program(s) on the companion CD-ROM



RAMDISK.ASM (Assembler listing)

Time routine

The TIME routine is fairly simple. For reading the clock, the routine reads the time from the memory locations of the clock, converts the time from BCD to binary format and then passes the time to the DOS buffer. For setting the time, the reverse occurs: The routine reads the time from the DOS buffer, converts the code from binary to BCD format and writes the BCD code into the memory locations of the clock. DOS uses the same format for indicating time as the clock: Hour, minute and seconds, each comprise one byte.

NOTE

The initialization routine INIT is more comprehensive than the CONDRV initialization routine and remains in memory after the end of execution even though it's no longer needed. We'll explain why this occurs in the section on the "INIT routine".

Date routine

The DATE routine is more complicated. While the clock stores day, month and year as one byte each, DOS encodes the date according to the number of days since January 1, 1980. This number must be converted into a date in the form of day, month and year as DOS writes the time and date. The opposite occurs when you call the Read function; the clock date must be converted into the number of days. Let's discuss how this is done.

The conversion routine begins with the year 1980. January 1, 1980 (called NUMDAYS from this point on) is equal to the value 0. The routine tests whether this year is less than the current year. If it is, the routine adds the number of days in this year to NUMDAYS, adding a day to compensate for each leap year. Then it increments the year and tests again for a smaller number than the current year. This loop repeats until it reaches the current year. The routine then computes the number of days in the current year's month of February and enters this month into a table that contains the number of days for each month.

In the next step, for every month less than the current month, the routine adds the number of days in this month to NUMDAYS. Once it reaches the current month, only the current days of the month are added to NUMDAYS. The end result is transferred to the DOS buffer and the routine terminates.

Converting to date format

Reverse this process to convert NUMDAYS into a date. The routine begins with the year 1980 and tests whether the number of days in this year is less than or equal to NUMDAYS. If this is the case, the year is incremented and the number of days in this year is subtracted from NUMDAYS. This loop is repeated until the number of days in a year is larger than NUMDAYS. The routine then computes the number of days in the current year's month of February and enters this month into the table of the months. January starts another loop, which tests whether the number of days in the current month is less than or equal to NUMDAYS. If this is the case, the month increments and the routine subtracts the number of days from NUMDAYS. If the number of days in a month is larger than NUMDAYS, the loop ends. NUMDAYS must only be incremented enough to give the day of the month and complete the date.

The routine then converts the date to BCD format and enters the date in the memory locations of the clock.

EXE Programs As Device Drivers

Over the past few years, more device drivers have been developed as EXE files. This means that you can either install the driver from a `DEVICE =` command in your `CONFIG.SYS` file or you could install it directly from the DOS system prompt or a batch file. Even DOS has started to use this kind of device driver. The `EMM386.EXE` driver available in DOS Version 5.0 is an example of this type of device driver.

These programs aren't true device drivers. Actually they are TSR programs that function as device drivers. Like other driver programs, they can be installed with the `CONFIG.SYS` file when the system boots, so they don't have to compete for memory with other TSR programs. These TSR drivers are character device drivers, which means that DOS doesn't assign drive letters to them. Their names are selected so they won't be confused with other files (e.g., `EMMXXX0` for EMS drivers). Also, since they aren't intended to replace normal device drivers, they can only use one driver function (function 00H for initialization). This function must be supported so that DOS can properly install the TSR program as a driver. Since you can't determine if one of the other driver functions might be called, such a function call should be indicated as an error from the status flag.

The `EXESYS.ASM` demonstration program shows how this works. This program was written to be installed as a device driver or called as a normal EXE file. DOS isn't restricted to the ".SYS" file extension for device drivers, even though most driver program files use it. DOS can load any EXE file as a device driver, as long as it consists of only one segment and starts at address 00H in this segment with the standard driver header.

This can be done easily by including the following instruction at the start of the segment:

```
org 0h
```

The driver header follows this instruction. When the driver is installed, DOS will then be able to obtain the offset addresses of the strategy and interrupt routines and call them as with any other driver. To run the program from the DOS command line, it must also have an entry point just like any other EXE file. This is easily done by inserting the name of a special initialization procedure (as would be found in a normal EXE program) after the `END` instruction at the end of the assembly language listing.

This way, the program will also be able to determine whether it was started as a device driver from `CONFIG.SYS` or from the DOS system prompt. The initialization procedure will only be called if the program is started from the command line. DOS will call the strategy and interrupt routines if the program is started from `CONFIG.SYS`. However, once you've installed a TSR program as a driver, why would you want to recall it? Doing this allows you to pass information to the driver by using command line parameters. This can easily be done with the `IOCTL` functions (DOS function 44H).

`EXESYS.ASM` contains all the basics for developing a driver that can be installed from `CONFIG.SYS` or called from the system prompt. This particular driver simply displays a message that indicates which method was used to start it. You can easily modify this program to perform more complicated driver operations.

You'll find the following program(s) on the companion CD-ROM



EXESYS.ASM (Assembler listing)

CD-ROMs

After its introduction in the audio world, the compact disk was introduced to the PC market. A CD-ROM drive and a PC form an interesting combination. The compact disk (CD) medium itself is read-only, but 660 megabytes of data can be stored in the form of text, graphics, etc. Many publications and references are currently available on CD-ROM, such as:

- Telephone directories
- Books in Print
- The Bible in various translations
- The English translation of Pravda

Also, maps, photographic libraries, public domain program collections and medical databases are available in CD-ROM format. New titles are being published daily in this growing market.

Why CD-ROM?

The CD-ROM has a clear advantage over the printed medium. Once captured and digitized, information can be processed by a computer in whatever form the user needs. The possibilities appear to be limitless, considering how easy it is to read and compare information. Another important advantage of CD-ROM is how easily data can be accessed. The user simply loads the driver software, presses a key or two and the information is displayed on the screen.

Currently, you can buy a PC-compatible CD-ROM player for \$100 to 300. These players are available as either external or internal devices.

Interfacing

The PC's hardware can be easily interfaced to a CD-ROM player. However, the software may encounter some problems. This is understandable, since DOS was never intended to support these devices. In this section we'll show you how a CD-ROM drive, using the proper drivers and utility programs, can be accessed like a read-only floppy disk drive.

We've mentioned the device drivers act as mediators between the disk operating system and the external devices, such as monitor, printer, disk drives and hard disks. DOS differentiates between block device drivers and character device drivers. As a mass storage device capable of reading information in a block mode, a CD-ROM drive would normally be added to the rest of the system through a block driver. Here's where the problem begins: DOS makes a number of assumptions about block devices, but a CD-ROM drive cannot meet this criteria.

Memory limitations

In versions of DOS up to and including Version 3.3, the biggest obstacle to interfacing with a block driver was the 32 megabyte limit imposed on every volume designated as a block device. The second biggest obstacle is the lack of a file allocation table (FAT) on a CD-ROM. Instead of the FAT, the CD-ROM contains a form of data table into which the starting addresses of the various subdirectories and files are recorded. However, DOS still requires a FAT that it can read during driver initialization.

A character device driver works better for implementing a CD-ROM device driver because DOS doesn't make assumptions about the structure of the devices connected through character drivers. However, even character drivers have problems communicating with a CD-ROM drive, because they transmit characters one at a time instead of in groups of characters. Another disadvantage is the need for a name (e.g., CON) instead of a device designation. DOS must first see the CD-ROM driver as a character driver to DOS to prevent read accesses to a nonexistent FAT. The CONFIG.SYS file supplies the name of the device during the system booting process.

Configuring the CD-ROM

The manufacturer usually includes CD-ROM driver software with the CD-ROM drive package. A driver of this type usually has a name such as SONY.SYS or HITACHI.SYS, depending on the manufacturer.

The CONFIG.SYS sequence that installs this driver can look something like this:

```
DEVICE=HITACHI.SYS /D:CDR1
```

The device driver selects the name CDR1 as the name of the CD-ROM drive.

After executing the initialization routine from DOS, the CD-ROM is treated as a block driver that has been enhanced with a few special functions supporting CD-ROMs. However, DOS still views the

You'll find the following program(s) on the companion CD-ROM



ATCLK.ASM (Assembler listing)

CD-ROM player as a character driver: DOS cannot view the CD-ROM's directory, nor can it directly access the files on the CD-ROM.

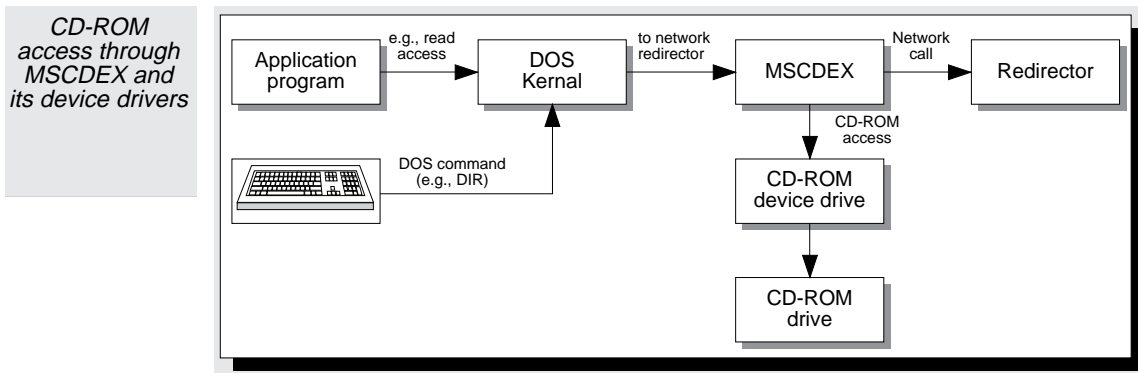
Driver software extensions

To overcome this obstacle, many CD-ROM players include a TSR (Terminate and Stay Resident) program, named MSCDEX (Microsoft CD-ROM Extension), in addition to the device driver software. This program must be called from within the AUTOEXEC.BAT file. The name of the CD driver can be passed to the program from the DOS prompt, as shown in the following example:

```
MSCDEX /D:CDR1
```

MSCDEX first opens this driver through the DOS OPEN function and provides it with a device designation. DOS assumes that MSCDEX is a device on a remote network, as supported by DOS in Version 3.1.

MSCDEX brings us closer to the solution, since DOS handles network devices as files containing more than 32 megabytes. These devices are accessed through redirection, rather than direct access from DOS. The resident portion of MSCDEX interfaces to the redirector and intercepts all calls to the redirector. If MSCDEX receives a call addressed to the CD-ROM drive, it adapts each instruction to a call applicable to the CD-ROM driver. This makes a perfect connection between DOS and the CD-ROM drive, while still allowing access to subdirectories and files at any time.



28

The Multiplexer (Interrupt 27H)

Several DOS commands operate as TSR programs. This means they become memory resident when called and remain in the background until needed. These include the PRINT, ASSIGN, SHARE, APPEND, DOSKEY, etc. commands.

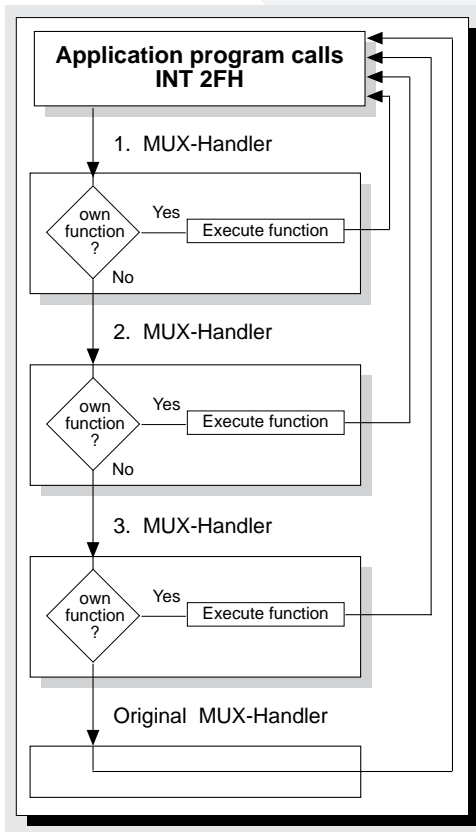
These programs use interrupt 27H so they can be accessed once they've become memory resident. This interrupt is called the multiplexer. We'll explain in this chapter how the multiplexer works and which programs use the multiplexer.

How The Multiplexer Operates

Interrupt 27H is known as the multiplexer because it acts as a communications interface for all TSR programs instead of only one DOS program. After installing a TSR program in memory with an initial function call, it's often necessary to set specific parameters for the TSR or remove the TSR program from memory using DOS commands. These commands then use the multiplexer interrupt to address the memory resident TSR program.

However, it's also possible for application programs to use these calls to modify various resident DOS programs or determine whether they are currently active. For example, a network program requiring SHARE can determine whether this DOS program was loaded. If SHARE wasn't loaded, its startup is aborted and an error message is displayed.

*The chain of
MUX handlers*



TSR programs must ensure that more than one program can use interrupt 27H simultaneously. This is done in the following way:

Before a program uses the multiplexer (or MUX), it must assign itself an eight bit identification number. This number is known as the MUX code. The MUX codes 00H to BFH are reserved for various DOS programs and 0CH to FFH are reserved for application programs.

When this type of TSR program is loaded, it must install an interrupt handler for the multiplexer interrupt, in addition to the other interrupt handlers it may need. So, the program must redirect the interrupt 27H to its own routine, which replaces the previous interrupt handler. When a program activates the multiplexer interrupt, the program's interrupt handler is executed.

First this interrupt handler must determine whether its own program or one of the other programs, which is tied to the multiplexer, is being addressed. It examines the AH register, which should contain the MUX code of the program that's being addressed.

If it finds its own MUX code in this register, the interrupt handler simply reads the remaining

processor registers that are important, executes its function, and returns to the caller program with an IRET machine language command.

However, if the MUX code refers to a different TSR program, the interrupt handler cannot terminate its call and return. Instead, it must call the previous interrupt handler that it displaced upon its installation in the multiplexer. This means that a TSR program must note the address of the interrupt handler currently installed in the multiplexer before it can install its own handler.

Since each multiplexer interrupt handler checks the MUX code in this way, and calls the previous handler if the code isn't its own, the chain of interrupt handlers is passed until one of the handlers recognizes the code as its own. This handler then executes the appropriate function, instead of calling its predecessor. At the end of this chain is the first handler that was installed by BIOS when the system was booted. This handler is preceded only by the IRET machine language instruction.

Competition between multiplex handlers

This system works until two multiplex handlers use the same MUX code. When this happens, the more recent handler along this chain will identify the code as its own, and call the specified function. The "older" handler cannot be accessed.

This problem won't occur in DOS programs because Microsoft ensures that none of these programs use the same MUX code. However, if you develop your own TSR programs, it's possible that you'll use a code that's already in use by another program. However, there is a way to determine this at the start of your program.

Normally, each TSR program is stopped to run a function for the installation check. This function should be executed if, at the multiplexer call, the AH register contains the MUX code of the particular program and the value 00H is found in the AL register. If one of the installed handlers recognizes its own code, it must place a value that doesn't equal 00H (generally FFH) in the AL register.

However, if none of the resident handlers recognize the code as their own, it will be passed along the chain of interrupt handlers until it reaches the first handler that was installed. This handler consists of only an IRET instruction. If this is the case, the AL register will be returned unchanged. So, if the AL register still contains 0 after this check, then the specified MUX code isn't being used by an installed handler.

In Chapter 35 we'll demonstrate how the multiplexer interrupt is used with TSR programs.

How DOS Programs Use The Multiplexer

In this section we'll discuss how the multiplexer is used by various DOS programs and which functions these programs provide. For application programs, many people believe that only the PRINT command, which enables you to print text in the background, is useful. However, you'll see that this isn't really true.

DOS commands which work with the multiplexer

The following table shows the DOS commands which work with the multiplexer:

DOS commands which work with the multiplexer			
MUX code	DOS command	MUX code	DOS command
01H	PRINT	48H	DOSKEY
06H	ASSIGN	4AH	DBLSPACE.BIN
10H	SHARE	4DH	KEYB
1AH	ANSI.SYS	B0H	GRAFTABL
43H	HIMEM	B7H	APPEND

The functions of these individual DOS commands are listed and briefly explained below. You'll find detailed descriptions of their input and output parameters in Appendix E on the companion CD-ROM.

PRINT *MUX code 01H*

The first DOS command to use the multiplexer was PRINT. The PRINT command uses this function to address its memory resident portion, to add files to the print queue list, to delete files from this list or to delete the list entirely.

Applications can also use this function when they need to execute long printing operations and don't want to monopolize the computer until the print operations are completed. Printers are normally slower than programs, which is why this function was introduced. To use PRINT, an application must "print" the desired output to a file and then add this file to the print queue. This can easily be done by using the PRINT function.

Function	Description
00H	Installation check
01H	Add file to queue
02H	Delete file from wait list
03H	Delete entire wait list
04H	Interrupt output and check status
05H	Resume output
06H	Check for printer

ASSIGN *MUX code 06H*

SHARE *MUX code 10H*

ANSI.SYS *MUX code 1AH*

GRAFTABL *MUX code B0H*

These commands support only function 00H, which is used for the installation check. If this function call returns the value FFH in register AL, that particular program has been installed. Refer to the previous table for the MUX code of each of these programs.

HIMEM.SYS *MUX code 43H*

The device driver HIMEM.SYS, which is responsible for managing memory according to the XMS standard, provides two multiplexer functions. The first function, 00H, supports the installation check. It returns the value 80H when HIMEM.SYS is installed. However, unlike other corresponding functions, it returns the value FFH. The second function represents the access point to this device driver, through which the various XMS functions are accessed. In this instance, however, the multiplexer simply enables access to HIMEM.SYS.

Function	Description
00H	Installation check
10H	Obtain access to XMS function calls

For more information about these functions and HIMEM.SYS, refer to Chapter 12.

DOSKEY (Version 5.0 and later) *MUX code 48H*

Since DOS 5.0, the memory resident program DOSKEY has permitted the storing and recalling of DOS command line entries, as well as the implementation of macros.

DOSKEY makes these functions available to applications, instead of only at the DOS command line. This is possible through multiplex functions. The MUX code of DOSKEY is 48H, and its two function numbers are 00H and 01H. As with other DOS commands, function 00H is used to check whether DOSKEY is installed. This function performs the usual installation check.

The second function can be used by an application program to ask DOSKEY to receive a command line from the user, in the same way as on the DOS command line. By using the cursor up and down keys, the user can move through previous DOS command line entries. New command lines entered with this function are also added to the list of entries stored by DOSKEY.

DBLSPACE.BIN *MUX code 4AH*

DoubleSpace, the DOS 6 online hard drive compressor, offers ten functions to disk utilities through the multiplexer interrupt. Unlike other MUX interfaces, 11H must always be given as the function number in register AL, with the subfunction number placed in register BX.

Function	Description	Function	Description
00H	Return version information	06H	Deactivate DoubleSpace drive
01H	Scan drive map	07H	Establish storage space
02H	Switch drive ID	08H	Return information about CVF file fragmentation
05H	Link compressed drive	09H	Scan for maximum number of compressed drives

See Chapter 34 for more information about DoubleSpace.

KEYB

MUX code ADH

The KEYB.COM keyboard device driver supplies a total of four multiplex functions. They can be used to check the current KEYB version number, the current code page and the current country flag.

Function	Description
80H	Return version number
81H	Determine current code page
82H	Set country flag
83H	Return country flag

APPEND

MUX code B7H

Of the MUX codes used by the various DOS commands, the code used by APPEND carries the highest value. Although the function number sequence is slightly unconventional, these multiplex functions supply an application with all the functions offered to the user at the DOS command line.

Function	Description
00H	Installation check
02H	Check compatibility to DOS Version 5.0
04H	Return directory list
06H	Check operating mode
07H	Set operating mode
11H	Establish conversion to complete filename



Network Programming

Over the last several years, networks have become an important part of using PCs. Networks were first used in large corporations, then in medium-sized companies, and now even in small private businesses. Most network systems can be purchased for several hundred dollars. As cheaper network cards become available, the cost of network systems will continue to decrease. Because networks have become more popular, the demand for programs that support networking is steadily increasing. In this chapter will present the basics of network programming. Although we can't discuss network programming in detail in this book, you should have enough information to create your first network programs.

Basics Of Network Programming

Networks can be used for various purposes. Usually they connect individual PCs, called workstations, to a more powerful server. The server is usually a fast PC, equipped with a large amount of memory and disk capacity. The server allows these workstations to use its resources and perform various tasks. One of the most common tasks is file sharing, which allows files, which are stored and managed by the server, to be accessed by individual workstations.

File sharing is usually performed with database files. Since these files are essential to a business' operation, they must be accessed by several employees. A good example of this is a mail order inventory. With a network, several employees, working on separate workstations, can directly access the inventory, which is stored in a central database.

Generally, the file server is also used to store programs that can be called from individual workstations. This saves disk space on the workstation and minimizes installation work. Obviously it's requires less work and time to install a program once on the file server, than to install it on every workstation. For example, Windows requires that certain files must be present on individual workstations, even if the main program has been installed on the file server. While the main Windows programs (WIN.COM, PROGMAN.EXE, etc.) can be located on the file server, each workstation needs its own WIN.INI and SYSTEM.INI files so each workstation can define its own environment.

However, even these files can be placed on the server by defining a private directory for each workstation or user. In this area, the private files of a given workstation can be stored without being modified by another workstation. A network operating system, which is the heart of any network system, manages these operations. Besides making its storage media accessible to other workstations, the network file server can act as a print or communications server, giving workstations access to a printer, a modem, or even another network. These capabilities can significantly reduce equipment expenses, since they allow one printer to be used by many workstations.

Since the appearance of OS/2, SQL servers have become more popular. These servers operate under OS/2, which allows the network stations to access SQL (Structured Query Language) databases. SQL is a program-based language for accessing relational databases. The SQL server not only manages the SQL databases, but also processes the queries directed to this database by individual workstations. For example, if a workstation queries a customer database for all addresses containing a zip code of 10000, the SQL server is responsible for screening and returning the appropriate records. This type of operation is often referred to as a remote procedure call, because the workstation is calling a procedure that's actually located in the SQL server.

The network operating system

Novell is the largest supplier of network operating systems. They offer different NetWare versions. However, several other network operating system manufacturers, such as IBM, 3Com, an Banyan Vines, also produce both NetWare-compatible and independent systems.

A network operating system always consists of a server program, which manages the file server, and a workstation program, which manages each workstation and allows it to address the server. This software is usually supplied in the form of a device driver or a TSR program. Workstations must be able to use DOS as usual and must access the file server like a normal floppy diskette or hard drive. Because of this, the software is embedded deeply into DOS, which enables it to control how DOS manages files.

The network software intercepts all file operations and passes only local operations, which access the workstation's own disk space, to DOS. For example, if the server in a certain network is identified by the device letter "S", the network software would intercept "S:LETTER.TXT" and address this call to the server. So, files that are located on the server can be addressed from a workstation in the same way as local DOS files.

However, this doesn't mean that a workstation user automatically has access to all the information stored on the server hard disk. Since this information could total 1 gigabyte in size, this could be an enormous amount of information. Instead, a network administrator defines which directories and files can be accessed by a user or workstation. These access rights are often tied to passwords that must be entered with a workstations user's login.

Since many users can access the server simultaneously, the server is rarely operated under DOS, which would be too slow to process numerous simultaneous access requests. So, Novell NetWare supplies its own operating system. For instance, this is a system that runs in protected mode on the server and that doesn't need or work with DOS in any way. The server hard disk is managed by a special Novell file system, instead of under DOS. Even if the server could be booted on a DOS diskette, the hard disk still couldn't be accessed because DOS is unable to recognize the foreign disk format.

However, you don't need this information for network programming under DOS, unless you plan on developing tools for Novell NetWare, which is a completely different process. This also applies to network hardware, which you also won't be dealing with internally, regardless of whether you're using a Token-Ring network, an Ethernet, or an Arc-Net. The function interfaces that facilitate access to the server and communication between workstations are more important.

Function interfaces

Three function interfaces have become standards for the PC. These are rudimentary DOS functions that were introduced with DOS Version 3.0. They allow several workstations to access server files. These functions are supported by all popular network operating systems. The IPX/SPX interface has been widely used because of Novell's large market share. This interface refers to the numerous functions that are available to a program running on a Novell NetWare workstation. These functions permit communication between workstations, which is vital for such things as E-mail or the remote operation of workstations.

NetBIOS interface, developed by IBM with its PC-LAN network operating system, works similarly. This interface is so widely used that Novell even offers a NetBIOS emulator for its Netware. This emulator transforms NetBIOS function calls into IPX/SPX functions. This is possible because both function interfaces were designed with the same operating principles. So, programs that have been written for NetBIOS can also run on Novell Netware systems. This is why we recommend NetBIOS for network programming, if you want to tackle such a task.

Peer to peer networking

Peer to peer networking has become very popular. With this type of networking, workstations can also act as servers. This technique is particularly suited for smaller operations, because purchasing a large and powerful server may be too expensive. In peer to peer networking, each workstation is equipped with a DOS supplement that allows the workstations to be accessed using standard DOS commands and file operations. Programs such as NetWare Lite by Novell and LANStatic by Microware use this technique.

Network Programming Under DOS

The two main DOS elements in network programming are DOS API functions and the SHARE program, which was introduced with DOS Version 3.0. This program implements the two DOS API functions, and, with their help, ensures that different programs don't interfere with one another while accessing files simultaneously. Often one program will run on several workstations simultaneously and store its files on the server, so these files can be accessed by the individual

workstations simultaneously. We'll use an example of a large health club to illustrate this. Several employees use individual workstations to manage the reservations for courts and equipment. These workstations are connected to a server via a network, where a central file contains all the court reservations for the coming days and weeks. The workstations use a single program, which manages court reservations and rentals. As we'll see below, the simultaneous actions of these employees create several problems for the network. First, however, let's take another look at the SHARE program.

Checking for SHARE

Because SHARE is needed to work with DOS networking functions, any network program you write should determine whether SHARE is present at program startup. Remember that SHARE is completely integrated in DOS Versions 4.0, 5.0, and higher, so you can omit this check if you're absolutely certain that your program won't be running on earlier versions of DOS.

The SHARE test consists of only an interrupt call. Simply call the multiplexer function 1000H, since this is where the resident portion of SHARE is located. If this function call returns the value FFH in the AL register, you know that SHARE has been installed. We talk about the multiplexer in Chapter 28.

Record locking

The most important aspect of simultaneous file access by more than one workstation is record locking. To explain this concept, let's return to our example of the health club. The employees take court reservations from members over the phone. On their workstations, the employees check the availability of the courts for a particular date and time. This information is provided by a data file. Since the data file's records are only read, problems don't occur.

However, when one of these records must be changed, conflicts can occur. For example, this happens when one of the employees makes a reservation for a particular court. Since this reservation isn't recorded in the data file, this court still appears as available. So, it's possible that another employee will also make a reservation for the same court at the same time and date for another member. If the software isn't fast enough to display the previous record change before the second employee makes his or her reservation, the first reservation will be overwritten by the second.

Record locking is designed to prevent these type of situation. Under this principle, a program must first take possession of a record and lock it before the record can be changed in any way. Once this is done, another workstation cannot modify the record.

In our example, this would work as follows: when the first employee finds the record for the available court, he or she presses a special function key that informs the program that this court should be reserved. The program then locks the record, so the necessary changes can be made to the record. Once this has occurred, the other employees cannot access the record from their workstations as long as it remains locked.

The program responds to this access attempt by displaying a message, which indicates that this court is currently being reserved. However, the first member may cancel his/her reservation immediately after placing it. So, the workstation that's requesting access to the record shouldn't give up immediately. Instead, it should repeatedly try to access the record until it becomes available or until a certain time limit is reached.

Once the record becomes available, the second employee can determine whether the court has already been reserved by another member or whether it's still available. However, this works only if the record is unlocked as quickly as possible after the change has been made.

The following guidelines apply to the shared use of files and records by one or more programs on numerous workstations:

- Records must be locked before they are modified, so other programs or workstations cannot access them during that time.
- Records must be released immediately after they are modified, so they are available to other programs or workstations.

SHARE, in conjunction with the network operating system, ensures that locked records cannot be accessed from other sources by displaying an error message when these function calls occur.

Although this process may sound complicated, it's actually very simple. Records can be locked with a single DOS function, 5CH. This function requires the correct file handle and the offset address and length of the segment to be locked. Since files are often larger than 64K, these parameters are considered 32 bit values and are therefore shared by two processor registers. The upper 16 bits of the start offset are written to the CX register and the lower 16 bits to the DX register. The segment length is handled in the same way, by registers SI and DI. So, although you're not dealing directly with records, locking them isn't very complicated. At a given record length, you can easily use the record number to calculate the correct start offset, and the segment length is already specified.

Since this function can also be used to unlock files, the AL register is used to indicate which operation the function should perform, either locking or unlocking the specified segment. It's also possible to lock more than one file segment with this function because many operations require more than one record to be locked at one time. This is necessary so other programs cannot access these records.

However, before a file or file segment can be locked, the file first must be opened. This is the only place you can obtain the handle the function needs to read from the BX register.

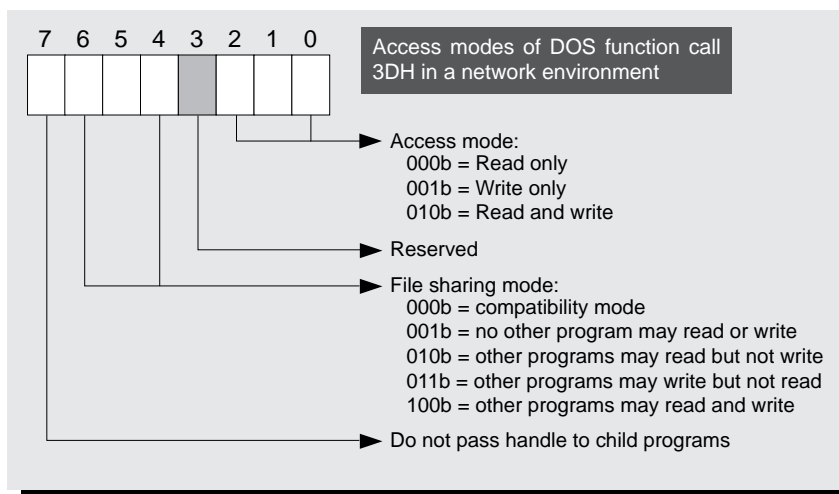
Register	Description
AH	5Ch
AL	Mode: 00 = Lock, 01 = Unlock
BX	File handle
CX:DX	Start offset in the source file as 32 Bit value
SI:DI	Segment (record) length as 32 Bit value
BU:	Parameters for function call 5Ch

File sharing

Besides record locking, SHARE lets DOS implement file locking, which prevents other programs and workstations from accessing a specific file. This option is activated by the DOS function 3DH, which is also used to open files in non-network situations. The only difference in network programming is that more information must be provided for this function in the AL register.

As the following illustration shows, bits 4 through 6 define the operations other programs are allowed to perform on the specified file in a network setting. So, you can determine if other programs can, for example, read but not write to your file or if access should be completely denied while your program is using the file.

Parameters for function call 3Dh	
Register	Description
AH	3DH
AL	Access mode *
DS:DX	Pointer to file name
* See illustration on the next page	



In this context, the lower three bits, which usually aren't significant in DOS programming, are important. These bits declare which operations a program will perform with a file upon opening it. A program using a READ file access mode when requesting access to an already open file, will be granted access to this file if the first program has released the file for READ operations through its file sharing mode.

So, in a network environment you shouldn't declare a READ and WRITE access mode if you only intend to read information from the file. Otherwise, you may be denied access to the file by a program that has already opened the file and denied WRITE access to other programs. If a file access request with a given access mode doesn't succeed, function 3DH will return error code 5 (access denied). This error code is also used by the functions for reading and writing files when they are denied access to a file.

You've probably noticed the "compatibility mode" among the file sharing modes listed in the diagram above. This mode serves the same purpose as mode 001b; it doesn't allow other programs to read from and write to the specified file. However, instead of calling the normal error code 5 (access denied), the compatibility code calls the DOS critical error handler when another program attempts to access the locked file. When such an error occurs, DOS displays the following message on the screen:

```
Illegal SHARE operation while reading drive C
(A)bort, (R)etry, (F)ail
```

Don't use this file sharing mode when other programs shouldn't access the locked file. Instead, use file sharing mode 001b.

An example of network programming

To demonstrate how to use DOS in network programming, we've developed two programs that use its file and record locking capabilities. You'll find the Pascal and C versions of these programs at the end of this chapter. A network isn't needed to use these programs; Windows 3.0 is sufficient. If you call SHARE prior to starting Windows 3.0, open several DOS boxes within Windows, and then start programs from these boxes, you can simulate any number of workstations.

You don't have to do this for the file locking programs (FLOCK.PAS and FLOCK.C) because these open a file twice from within the same program. This is the same as trying to open the file from a different program.

First, look at the two modules NETFILE.PAS and NETFILE.C, which provide the essential file access procedures for the two sample programs. Both of the programs use the same functions (or procedures) and constants. The difference between the two lies in the way the file, that should be accessed, is passed to these two programs. A list of available functions and procedures is listed in the table to the right.

Although the Pascal version has been programmed as a UNIT, you can integrate the C version into a program using #INCLUDE. So, compiling the module and defining a make or project file isn't necessary.

Similar to the Pascal commands ASSIGN, READ, and WRITE, the functions and procedures of the Pascal version accept a normal, standardized file variable. However, in a network environment, these commands cannot be used for accessing this type of a file, because they execute as soon as a file isn't open to read or write access. Because of this, special procedures for reading from and writing to files have been defined in the NETFILEP unit. However, using Turbo Pascal file variables is still useful because they provide the procedures with, for example, the length of individual records within the file. The file handle and access mode, which are otherwise entered by the REWRITE Pascal procedure, are also stored here.

In the C version, the file is represented by the following data structure, which should be passed, in the form of a variable, to the various functions in the NETFILEC.C module:

Procedures and functions provided by modules NETFILE.PAS and NETFILE.C.	
Name	Description
ShareInst	Checks whether SHARE is installed
NetErrorMsg	Returns an error message text
NetRewrite	Creates a file
NetReset	Opens a specific file
NetClose	Closes a file
NetLock	Locks records
NetUnLock	Unlocks locked records
Is_NetWriteO	Checks whether file may be written to
Is_NetReadO	Checks whether file may be read from
Is_NetOpen	Checks whether file is open
NetWrite	Writes to file
NetRead	Reads from file
NetSeek	Sets data pointer

```
typedef struct { unsigned int Handle,    /* File handle */
                RecS,                /* Record size */
                Mode;                /* Access mode */
} NFILE;
```

The FLOCKP.PAS and FLOCKC.C programs illustrate how files are locked. First they ask the user for the access and file sharing modes for the two files that should be opened. However, the program actually opens only one file, named either FLOCKC.DAT or FLOCKP.DAT, depending on the program used. This file is then opened by two OPEN statements, so it can be addressed simultaneously over two handles. This simulates a network environment in which two programs are accessing the file simultaneously.

Upon opening these two handles, the DOS error messages are displayed on the screen. These handles are also used to execute write and read operations, the status of which is also displayed on the screen. After the handles have been opened, the text "AAAA" is written to the first handle, and the text "BBBB" is written to the second handle.

The file content is read through both handles in the next program step, and is then displayed on the screen. If the write access over the second handle was successful, the text "BBBB" will be displayed through both handles. This means the first text has been overwritten, assuming the first write attempt actually was successful. If both write access attempts failed, the file will be empty and two empty strings will appear on your screen. The last program step finally closes both handles.

These two programs are helpful because they can be used to test all different combinations of access and file sharing modes. However, some of them don't make sense and may lead to rather strange results. However, these results are similar to what sometimes occurs in a network environment.

Unlike the two FLOCK programs, the RELOCK programs (RELOCKP.PAS and RELOCKC.C) must either run on two network computers or within two separate DOS boxes under Windows. These programs lock specified records of the RELOCKP.DAT (Pascal version) or RELOCKC.DAT (C version) file. This file is created at the beginning of the program and is equipped with a total of 10 records, each containing 160 bytes. These records are assigned specific ASCII codes. The first record receives the ASCII code of the capital letter A, the second record the capital letter B, and the third the capital letter C, etc.

Both programs allow you to access different records, to read from and write to them, and, most importantly, to lock them. Currently locked records are indicated at the right side of the screen.

It may be interesting to lock several records from within the first program and then attempt to access these from the second program. DOS won't permit this access and will return error code 5, which displays an error message on your screen.

You'll find the following program(s) on the companion CD-ROM



NETFILE.PAS (Pascal listing)
NETFILEC.C (C listing)

You'll find the following program(s) on the companion CD-ROM



FLOCKP.PAS (Pascal listing)
FLOCKC.C (C listing)



DOS And Windows

Today anyone who produces DOS applications must expect that these applications will eventually be executed through Windows. Although Windows successfully simulates the DOS environment, there are certain things that DOS programs cannot do under Windows. So, if you're not careful, particularly with disk utilities, Windows may crash. TSR programs that access the DOS core deeply in order to shield themselves from other TSR programs will also cause problems.

In this chapter we'll discuss how a DOS application can detect the presence of Windows.

Sensing Windows

No single DOS function is available to indicate whether Windows is active, which version is active and in which mode Windows is running. Multiple, unrelated functions must be called to provide this information.

The multiplex interrupt (interrupt 2FH) plays an important role in checking for Windows. Many resident DOS applications and utilities, such as SHARE and PRINT, use this interrupt to make their services available to other applications. Windows also assigns a few functions to this interrupt.

As the flowchart on the next page indicates, we begin by calling multiplex function 1600H, which is provided by Windows to specify the Windows operating mode. If another function wasn't linked to this function number in the multiplex interrupt, then the register settings will remain unchanged by the function call, and the AL register will remain set to 00H. This means that Windows 3.x is inactive. However, other tests must be performed to determine whether a different version of Windows is active.

If the AL register contains either 01H or FFH after the function call, Windows 2.x is active. Although the exact version is still unknown, this value indicates a running version of Windows.

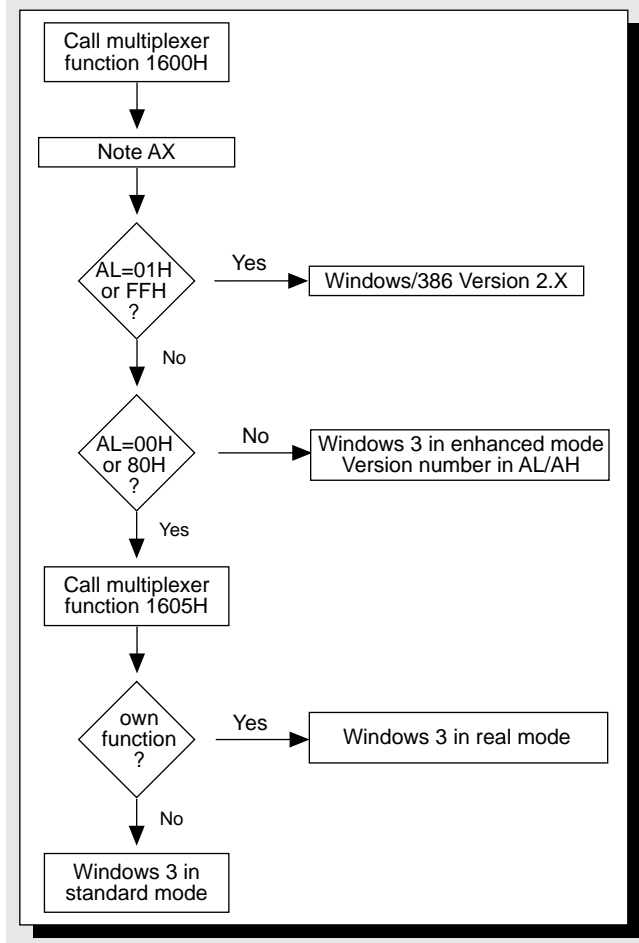
If the AL register contains 80H, Windows 3.x is active in a mode other than enhanced mode.

If the AL register contains values other than 00H, 01H, 80H, or FFH, Windows 3.x or Windows 95 is running in enhanced mode. The AL register value represents the major version number of the current Windows system, while the AH register value represents the minor version number.

Additional tests are needed to gather more information if the function call returned 00H or 80H. To determine whether Windows 3.x is active, use function 4680H, which links Windows 3.x to the multiplex interrupt. If this function returns the value 80H in the AL register, the function isn't even implemented and you can assume that no version of Windows is active.

The following flowchart illustrates how the Windows check should be performed. However, if you receive the value 00H, Windows 3 will be active. From there, all you need to do is determine the Windows operating mode. It can be running in either the standard or the real mode.

Algorithm for checking the Windows version and the operating mode



Function 1605H will help us determine the operating mode. This function switches Windows to standard mode from its current operating mode. If Windows is already in standard mode when function 1605H is called, this function will fail, placing 00H in the CX register to indicate this failure. Function 1606H switches Windows from standard mode into real mode. If Windows is already in real mode when function 1606H is called, this function will fail, placing 00H in the CX register to indicate this failure. Either function indicates the Windows 3.x operating mode, thus completing the test.

Sample programs

The WINDAB.BAS, WINDAP.PAS and WINDAC.C programs in BASIC, C, and Pascal implement the algorithm we've described. Each program contains a routine called WINDOWS, which returns constants named NO_WIN, WIN_386_X, WIN_REAL, WIN_STANDARD, or WIN_ENHANCED, depending on Windows status when the program is run. The programs will even tell you the exact version number if Windows 3.x is running in enhanced mode. For this, the function requires two integer variables (or in the case of the C implementation, the addresses of these variables) for storing the Windows version number.

You'll find the following program(s) on the companion CD-ROM



WINDAB.BAS (BASIC listing)
WINDAP.PAS (Pascal listing)
WINDAC.C (C listing)

31

Maintaining Compatibility

We discuss three ways to access PC hardware in this book. You can either access available DOS or BIOS functions or develop new functions and routines for direct hardware control. Although this doesn't provide any advantages in mass storage device and keyboard access, special routines for screen display are often much faster and more efficient than BIOS and DOS routines that perform the same task.

We recommend using DOS functions for compatibility. To develop programs that run without problems on virtually any DOS computer, you must follow some rules for DOS function calls. To develop programs, under the current DOS versions, that should execute without problems under future versions of DOS, follow these suggestions:

- Use only DOS functions for screen and hardware access. Don't use BIOS or other hardware dependent functions.
- Display error messages on the standard error device (handle 2).
- Use Version 2 UNIX-compatible handle functions for file access. This ensures compatibility with future versions of DOS. If you must use old FCB functions for file or directory access (e.g., for special attributes), be sure no FCBs, which are already open, are opened and no FCBs, which are already closed, are closed.
- Check the DOS version number at the beginning of the program and end the program with an error message if it cannot execute under this version.
- Store as many constants as needed for program execution (e.g., the paths of programs and files to be loaded) within the environment block. Access these values from the environment block within the program.
- Release all memory not required by the program using the DOS functions. (This is especially important when working with COM programs.)
- If you need additional memory, request it using the proper DOS functions.
- Use the available DOS functions for interrupt vectors; don't directly access interrupt vectors. To change the contents of various interrupt vectors within a program, first save the old contents and restore them before the end of the program.
- Call one of the DOS functions (31H or 4CH) before the end of the program to pass a value to the calling program to signal whether the program was executed correctly. Avoid using the other functions for ending a program (interrupt 20H and function 0 of interrupt 21H).
- Use function 59H of interrupt 21H (available in DOS Versions 3.0 and higher) to localize error sources.

The table on the following page is an overview of the older DOS functions you should avoid and their replacements:

Old		New	
00H	End program	4CH	End Process
0FH	Open file	3DH	Open Handle
10H	Close file	3EH	Close handle
11H	Find first entry	4EH	Find first entry
12H	Find next entry	4FH	Find next entry
13H	Erase file	41H	Erase directory entry
14H	Sequential read	3FH	Read (through handle)
15H	Sequential writ	40H	Write (through handle)
16H	Created file	3CH 5AH 5BH	Created handle or Created temporary file or Created new file
17H	Rename file	56H	Rename directory entry
21H	Random access read	3FH	Read (through handle)
22H	Random access write	40H	Write (through handle)
23H	Sense file size	42H	Move file pointer
24H	Set data set number	42H	Move file pointer
26H	Create new PSP	4BH	Load and execute from file
27H	Random access read	3FH	Read (through handle)
28H	Random access write	40H	Write (through handle)

If you follow all these suggestions, your programs will execute on other computers and under future DOS versions with little or no modifications.



Undocumented DOS Structures

Many reference books discuss undocumented information about DOS. Occasionally, we've found some DOS structures still undocumented, such as several DOS variables that contain extremely important information. In this chapter, we'll discuss some of these undocumented structures.

Documented And Undocumented Structures

DOS manages the operating storage media (RAM and mass storage) and programs that use multiple data structures. Some of these structures are thoroughly documented and have already been described in this book. These structures include:

- Program Segment Prefix (PSP), which precedes every program in memory
- File Control Blocks (FCBs), which control file access
- Memory Control Blocks (MCBs), which control RAM
- Structures in the header of a device driver
- Environment blocks, which contain information strings about every program in memory
- The many structures that DOS keeps in mass storage (boot sector, File Allocation Table [FAT], root directory, etc.)

There are also several undocumented structures. Until recently, only a few people knew of these structures because most technical manuals about DOS didn't describe them. The authors of many of these manuals believed these structures weren't needed for programming, and their coding would change in future versions of DOS. Actually, certain kinds of programming do depend on these structures and some applications couldn't be created without them.

Floppy disk and hard drive management utilities extensively use these undocumented structures. For example, if you examine Norton Utilities with a debugging application, you'd see how much this program accesses these structures.

A minor change in these undocumented structures occurred between DOS Version 3.3 and Version 4.0, but this is the first change since the introduction of DOS Version 2.0 in 1983. So, you probably won't find altered coding in the undocumented structures of subsequent DOS versions.

Knowing about these structures can be very useful when you're programming certain applications.

The DOS Info Block (DIB)

The DOS Info Block (DIB) is the key to accessing the most important DOS structures. This block contains pointers to several DOS structures and to other information. The DIB is useful to a program only if its address in memory is known. This address isn't in a fixed memory location, and cannot be obtained with any of the documented functions of DOS interrupt 21H. However, the undocumented function 52H can help locate this address. Calling function 52H returns the address of the DOS Info Block to the ES:BX register pair.

Unlike other DOS functions that retrieve pointers to a structure or data area, the contents of the ES:BX register pair point to the second, instead of the first, field within the DIB after the function call.

DIB structure

The first field in the DIB contains a pointer to the Memory Control Block (MCB) of the first allocated memory area.

Drive Parameter Block (DPB)

The pointer in the second field of the DIB provides access to information that cannot be accessed in any other way. It points to the first Drive Parameter Block (DPB), which is a structure that DOS creates for all mass storage devices (floppy diskettes, hard drives, tape drives, etc.).

The first field of the DPB indicates the device to which the block belongs. 0 represents drive A, 1 represents B, 2 represents C, etc. The second field specifies the number of the subunit. To understand the meaning of this field, remember that access to the individual devices occurs through the device driver. DOS doesn't perform direct access to a disk drive or hard drive. So, DOS doesn't have to deal with the physical characteristics of a mass storage device. Instead, DOS calls a device driver, which acts as mediator between DOS and hardware.

Obviously, not every device has a separate device driver, since one device driver can support many devices. For example, the device driver built into DOS manages the floppy disk drives and the first available hard drive.

Since DOS configures a DPB for each device, a hard drive system automatically has 3 DPBs available. (A DPB is always configured for floppy disk drive B, even if only one floppy disk drive is actually available.)

Each device receives a number between 0 and the total number of devices minus 1, to help each driver identify the devices it manages. This is the number found in the subunit field.

The next field lists the number of bytes per sector. Under DOS, this is usually 512. After this is the interleave factor, which provides the number of logical sectors displaced by physical sectors when the medium is formatted. This value can be 1 for floppy disk drives, 6 for the XT hard drive, and 3 for the AT hard drive. For floppy disk drives, this field can also have the value FEH if the disk in the drive hasn't been accessed. The value FEH indicates the interleave factor is currently unknown.

Several other fields are related to these two fields. We mentioned these fields when we discussed managing mass storage devices through DOS. Among other things, these fields describe the status and the size of the structures DOS created to manage mass storage devices. A pointer to the header of the device driver is located within these fields. DOS uses this pointer when accessing the device. Additional information can be obtained with this pointer since, for example, the driver attribute is listed in the header of the device driver.

DOS Info Block (DIB) structure		
Addr.	Contents	Type
-04H	Pointer to MCB	1 ptr
ES:BX	Pointer to first Drive Parameter Block (DPB)	1 ptr
+04H	Pointer to last DOS buffer	1 ptr
+08H	Pointer to clock driver (CLOCK)	1 ptr
+0CH	Pointer to console driver (CON)	1 ptr
+10H	Maximum sector length (based on connected drives)	1 word
+12H	Pointer to first DOS buffer	1 ptr
+16H	Pointer to path table	1 ptr
+1AH	Pointer to System File Table (SFT)	1 ptr
Length: 1EH (30) bytes		

Drive Parameter Block (DPB) structure		
Addr.	Contents	Type
+00H	Drive number or character (0 = A, 1 = B, etc.)	1 byte
+01H	Sub-unit of device driver for drive	1 byte
+02H	Bytes per sector	1 word
+04H	Interleave factor	1 byte
+05H	Sectors per cluster	1 byte
+06H	Reserved sectors (for boot sector)	1 word
+08H	Number of File Allocation Tables (FATs)	1 byte
+09H	Number of entries in root directory	1 word
+0BH	First occupied sector	1 word
+0DH	Last occupied cluster	1 word
+0FH	Sectors per FAT	1 byte
+10H	First data sector	1 word
+12H	Pointer to header (corresponding device driver)	1 ptr
+16H	Media descriptor	1 byte
+17H	Used flag (0FFH = Device not yet enabled)	1 byte
+18H	Pointer to next DPB (xxxx:FFFF = last DPB)	1 ptr
Length: 1CH (28) bytes		

Following this field is the media descriptor to which the Used flag is connected. As long as the device hasn't been accessed, this flag contains the value 0FFH. After the first access, it changes to 0 and remains unchanged until a system reset.

The DPB ends with a pointer that establishes communication with the next DPB. Since every DPB defines its end with such a pointer, a kind of chain is created, through which all DPBs can be reached. To signal the end of the chain, the offset address of this pointer in the last DPB contains the value 0FFFFH.

DPB access

When a program needs the information within the DPB, there are many ways to find the address of the desired DPB. One method is to follow the chain previously described by first determining the address of the DIB. This gives you the pointer to the first DPB, from which you can follow the chain until you reach the desired DPB.

However, there's a better method, which isn't as susceptible to changes within the DIB. This method involves two undocumented DOS functions, 1FH and 32H functions. Although these functions have been included in DOS since Version 2.0, they weren't documented by Microsoft. When called, both return a pointer to a DPB to the DS:BX register pair. While function 1FH always delivers a pointer to the DPB of the current disk drive, the address delivered by function 32H refers to the device whose number is passed to the function in the DL register at the time it's called (0 represents the current drive, 1 is drive A, 2 drive B etc.). Function 32H is much more flexible than function 1FH.

Using 1FH and 32H to access the various DPBs is also useful because it forces DOS to retrieve other information, such as the interleave factor and the media descriptor byte, which is determined for the disk drive only after the first access. If you get to the DPB through the pointer in the DIB block, the various fields may not have been initialized, and could contain the wrong values.

The DOS buffer

Besides the pointer to the first DPB, the DIB also contains the pointer to the first DOS buffer at address 12H. These DOS buffers store individual sectors, so the sectors don't have to be repeatedly loaded from disk. The DOS buffers are most effective when they're used to store disk sectors that are frequently needed by the currently running program. Besides the FAT, these sectors include the root directory and its subdirectories. The number of buffers can be defined by the user in the CONFIG.SYS file. If this number exceeds those needed for the FAT, root directory, and subdirectories, normal sectors can also be temporarily stored here. This is done so if they are called again in the near future, they can be taken directly from the buffer.

The individual sectors are linked together. This enables DOS to quickly check each buffer for the desired sector with each read operation.

DOS buffer structure		
Addr.	Contents	Type
+00H	Pointer to next DOS buffer	1 byte
+04H	Drive number (0 = A, 1 = B, etc.)	1 byte
+05H	Flags	1 byte
+06H	Sector number	1 word
+08H	Reserved	2 bytes
+0AH	Contents of buffered sector	512 bytes
Length: 210H (528) bytes		

As with DPBs, this occurs with the help of a pointer that appears at the start of every buffer. Also, the last buffer is reached when the offset address of the pointer contains the value 0FFFFH. After the field linking one buffer to the next is the number of the drive where the buffered sector originates. The value is 0 for drive A, 1 for B, 2 for C, etc. Besides the drive number, the identification of a sector requires a sector number. This is located beginning at position 06H in the DOS buffer. The last field in the buffer header stores a pointer to the corresponding DPB, so DOS can obtain information about the device that loaded the buffered sector. Although this is the last field in the header of the DOS buffer, the buffered sector doesn't end immediately after this field. There are two more bytes which follow. The reason for this is the DOS code is written in machine language. So, when working with memory blocks, it's most efficient to have the buffered sector begin with an address that is divisible by 16.

The path table

The header of the DOS buffer isn't the last place we encounter the DPB. It appears again in the path table, which starts at address 16H in the DIB. This table contains the current path for each drive as well as a pointer to its DPB.

*Memory dump
of the path table
contents*

```

  0 1 2 3 4 5 6 7 8 9 A B C D E F
0000: 41 3A 5C 43 41 43 48 45-00 00 00 00 00 00 00 00 00 A:\CACHE.....
0010: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0040: 00 00 00 00 40 20 74 80-02 27 03 FF FF FF FF 02 .....@ t.'.....
0050: 00 42 3A 5C 00 00 00 00-00 00 00 00 00 00 00 00 00 .B:\.....
0060: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0070: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0080: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0090: 00 00 00 00 40 40 74 80-02 00 00 FF FF FF FF .....@t.....
00A0: 02 00 43 3A 5C 54 43 5C-42 41 55 53 5C 41 53 4D ..C:\TC\BAUS\ASM
00B0: 5C 48 45 52 43 4D 4F 4E-4F 00 00 00 00 00 00 00 00 \HERCMONO.....
00C0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
00D0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
00E0: 00 00 00 00 00 00 40 60-74 80 02 65 05 FF FF FF .....@`t.e....
00F0: FF 02 00 44 3A 5C 4D 53-43 5C 42 49 4E 00 00 00 ...D:\MSC\BIN...
0100: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0110: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0120: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
0130: 00 00 00 00 00 00 00 00-40 00 00 80 0D 17 00 FF FF .....@.....
0140: FF FF 02 00

```

As long as the LASTDRIVE command is in the system's configuration file, the table will have entries for drives A through the one specified by LASTDRIVE. If this command is missing, however, the table will have entries only for each device supported by the installed device driver. If you change the entries in this table, you can divert one drive to another. The JOIN and SUBST DOS commands also use this by manipulating the path table entry of the drive to be diverted.

33

An Introduction To DOS 6.x

In 1993, Microsoft introduced the sixth version of DOS. Many users were surprised by how few of the promised features actually appeared in this version. For example, multitasking wasn't included. Instead, DoubleSpace and third-party memory support were added. In our opinion, DoubleSpace is an important new feature of DOS 6.x.

In this chapter, we'll discuss how DoubleSpace works and how it organizes information on a drive. We'll also discuss the DoubleSpace user interface. Then we'll explain the compression/decompression algorithms on which DoubleSpace is based, and show how you can use data compression and decompression capabilities in your own applications.

DoubleSpace

MS-DOS 6 includes an online disk compressor called DoubleSpace. Since similar products existed prior to DOS 6 (e.g., Stacker), DOS 6 provides for optimal integration of these components into the total system. Although the maximum performance is decreased by 10 percent, the hard drive capacity is almost doubled. First, we'll examine some basic concepts of disk compression, then we'll look at the internal workings of DoubleSpace. You'll learn how DoubleSpace manages the compressed data and what DoubleSpace has to offer.

Data compression

There's nothing mysterious about data compression. Compressing files and directories produces an (apparent) expansion of available hard drive space. All compression programs, such as LHARC, Stacker, and DoubleSpace, condense data according to one of the following algorithms:

- Run Length Encoding (RLE)
- Huffman Coding
- Lempel-Ziv Compression, also known as LZW (Lempel-Ziv-Welsh).

There are several variations of these algorithms. However, each is based on a specific principle of data compression. All three processes have advantages and disadvantages regarding the amount of compression that's possible and the system resources needed for compression and decompression. Obviously resource utilization also affects the speed of compression and decompression, which, in turn, affects file access performance. However, an online compressor such as DoubleSpace provides a balanced relationship between degree of compression and performance.

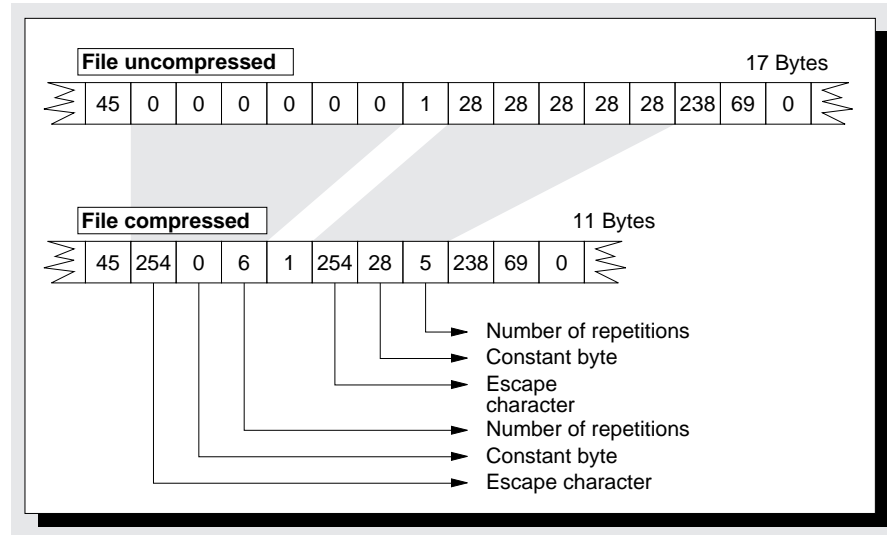
The following descriptions of the various compression techniques should help explain why the developers at Microsoft decided to use the Lempel-Ziv process.

Run Length Encoding

Run Length Encoding is the simplest form of data compression. When compressing a file containing sequences of identical bytes, only the first byte of each sequence is stored, followed by the number of repetitions. These two bytes must be preceded by an ESC character. During unpacking the ESC character indicates that a repetitive sequence was encoded in this particular location. Since the Escape character can also occur as a "normal" character in the file, additional precautions must be taken to ensure that this character isn't later interpreted as the start of a repetitive sequence. This doesn't affect the basic mechanism of Run Length Encoding, however.

The following illustration shows how a file is compressed using this method:

Compression with Run Length Encoding



This type of data compression is especially suited to files containing numerous sequences of identical bytes, such as graphic files. Usually the greater the frequency of such sequences and the longer they are, the higher the degree of compression attained. The RLE process is relatively easy to implement, yet for large groups of differentiated files it yields the poorest results of all three algorithms. This is because it affects only constant byte sequences and leaves all other characters untouched.

Huffman coding

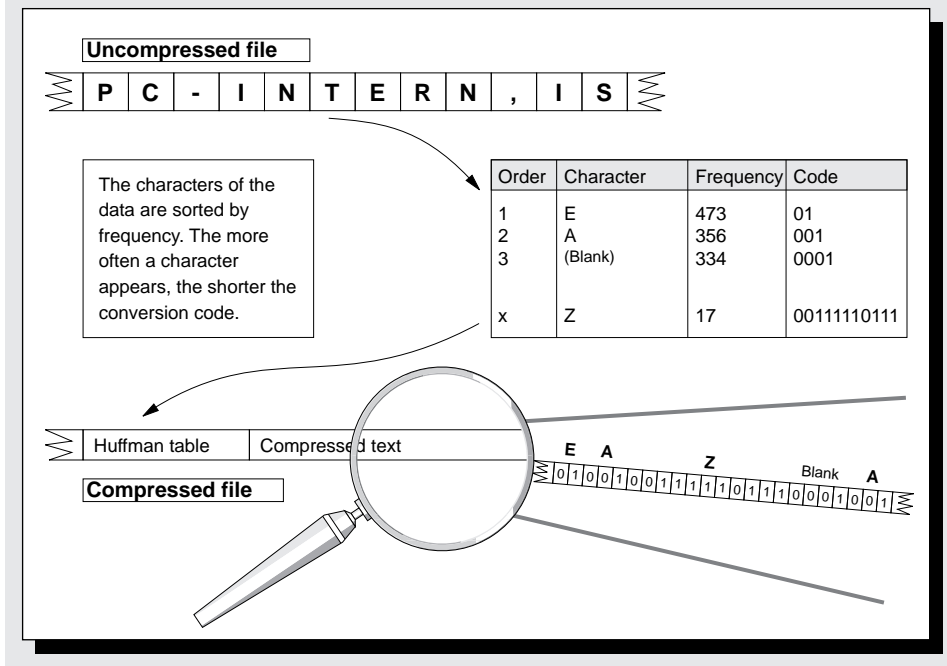
Huffman coding, named for the French mathematician Huffman, deviates from the commonly used standard in electronic data processing. Usually each character in a text or a file is represented by a constant number of bits (usually 8). However, with the Huffman method, the individual characters receive variable lengths.

The Huffman algorithm takes the file to be compressed and analyzes the frequencies with which various characters occur. Depending on these frequencies, the characters are then portrayed as bit sequences of varying lengths. The more frequently a character appears in the file, the shorter the bit sequence. As a result, some characters have bit lengths greater than eight. However, in the middle range of frequencies, lengths remain below eight, while the more frequently occurring characters are coded in far fewer than eight bits.

The actual process, once these bit sequences have been established and converted, is based on the creation of a binary tree, which contains the different characters and their frequencies. Since this is a relatively complicated procedure, we won't discuss it in detail here. The result of this process is that two bits are still needed for coding the most common characters, while longer bit sequences are needed for the less common characters. These sequences are usually longer than the original eight bits. However, a noticeable compression occurs relating to the common characters.

When comparing degrees of compression, the Huffman process is always superior to Run Length Encoding and in most cases superior to the LZ process. However, this depends on the contents of the file. The difference is smaller with the LZ process than with RLE. The Huffman process's disadvantage, as compared with the LZ process, lies in its complex conversions and the relative slowness of subsequent decompression. The file contents must be interpreted as bit sequences of varying lengths instead of as a group of 8-bit characters.

Structure of
video RAM in
EGA and VGA
16 color
graphics modes



Lempel-Ziv process

Although the LZ process produces compression rates comparable to those of Huffman coding, it is much simpler and even has some similarities to Run Length Encoding. The focal point of this process is a search for repetitive sequences within a file. While Run Length Encoding looks only for sequences of the same byte, the LZ process checks the entire text. For example, suppose the word "Miller" occurs in a file. The LZ algorithm searches for this word in the stored text. If it's found, a "match tag" is written into the file in place of the word. As a type of reverse-direction offset, the match tag indicates how many characters back one must go from the current location to the referenced character sequence, and how many characters should be taken from that point. In addition, the LZ process also incorporates Run Length Encoding for sequences of identical characters. The repeated character is written once to the file at its respective position and immediately after it's placed a match tag. Here the offset is 1 and the number equals the number of repetitions.

Of course this process is useful only if the match tag doesn't require more space than the text to which it refers. The version of the algorithm incorporated into DoubleSpace considers this and, depending on the type of file, achieves at least the following rates of compression (see the table on the right). In this way, with a balanced mix of files, DoubleSpace can virtually double available hard drive capacity.

File type	Comp. ratio
Program files	1.4:1
Texts, spreadsheets, databases	2:1
Graphics and other highly redundant files	3:1 or more

Compressed Volume File (CVF)

To optimize the available hard drive capacity, DoubleSpace uses a CVF (Compressed Volume File). Usually this file, whose fixed size is determined when DoubleSpace is installed, occupies the majority of the storage space. The CVF includes both compressed files and all data structures needed for file upkeep. With DoubleSpace, you can have more than one CVF file within a system. Each of these files is treated as a separate drive with its own drive identifier (i.e., virtual drives) and can occupy up to 512 Meg on the (uncompressed) hard drive. So, after compression, up to 1 Gigabyte or more of files can be stored. The name "DBLSPACE.nnnh" is always assigned to this file; nnn is a sequential number. In converting an existing drive to a DoubleSpace drive, DoubleSpace always assigns the number "000" to the CVF file that is generated. Therefore, the file's name becomes "DBLSPACE.000."

When you start the system, this file isn't visible on drive C:. This occurs because what DOS states is drive C: is actually the contents of the CVF file; the original files from drive C: are stored as compressed files within the CVF file. However, the CVF file is accessible because the original drive C: now has drive identifier H:. When you look in the main directory of this drive, you'll see the file "DBLSPACE.000." You can also assign a device ID other than H:, as long you specifically define it during hard drive conversion.

While DoubleSpace sets up a slightly modified FAT system within the CVF file, the original drive (now H:) remains unchanged. Its structure and any files subsequently placed on it won't change. In particular, DoubleSpace leaves the DOS system files on the drive so the system can boot up as before from the hard drive. Actually, DoubleSpace is activated relatively late in the boot sequence. So, it isn't immediately available when the computer is switched on. The system first boots once using the normal FAT on drive C:. From the perspective of the C: drive, the CVF file looks like a completely normal file. To display the contents of drive H:, use the DIR H: /A:HS command. The following files will appear:

*Use the DIR H:
/A:HS command
to display the
contents of drive
H:*

```
Volume in Drive H is HOST_FOR_C
Volume Serial Number is 1AAB-734F
Directory of H:\

IO          SYS          40,767   03-10-93   6:00a
MSDOS       SYS          38,186   03-10-93   6:00a
386SPART    PAR       16,769,024  08-12-93   3:17p
DBLSPACE    BIN          51,288   03-10-93   6:00a
DBLSPACE    INI           91    06-30-93  10:37a
DBLSPACE 000 295,322,624  08-12-93   3:16p
6 File(s)  312,221,980 bytes
111,099,904 bytes free
```

In addition to the DOS system files IO.SYS and MSDOS.SYS, and the CVF file DBLSPACE.000, three other files exist on this drive. One is the permanent Windows swap file (if installed), which for performance reasons should not be stored within a compressed drive. There are also two other DBLSPACE files required to set up the DoubleSpace drive at system startup (more on this later). The following table summarizes the various DoubleSpace files:

Filename	Purpose
DBLSPACE.BIN	Enables access to compressed drives. This file is loaded while booting and executes even before DOS begins processing the CONFIG.SYS file.
DBLSPACE.EXE	Sets up and maintains DoubleSpace drives.
DBLSPACE.HLP	Help file for DBLSPACE.EXE.
DBLSPACE.INF	Stores the DoubleSpace configuration.
DBLSPACE.INI	DoubleSpace configuration file for Windows.
DBLSPACE.WIN	Used only during installation to record information about the current Windows system.
DBLSPACE.SYS	Device driver that determines the final location of the DBLSPACE.BIN program code and moves the code into upper memory when requested.
DBLSPACE.xxx	DoubleSpace CVF file containing a compressed drive.
DBLWIN.HLP	Help file for running DoubleSpace under Windows.

CVF file structure

To understand the structure of a CVF file, you must remember two requirements that existed when DoubleSpace was being developed and significantly influenced its development:

1. A DoubleSpace drive must appear as a normal DOS drive for utility programs (i.e., Norton's SpeedDisk and DirectorySort).
2. The files stored on a DoubleSpace drive must always be compressed in blocks of fixed size instead of compressed in their entirety.

The first requirement is related to compatibility, which is extremely important in DOS. So existing application programs and disk utilities can continue using DOS file access commands (especially interrupts 25H and 26H) to access DoubleSpace drives as well, the original FAT structure of such a drive must remain as close to intact as possible. Later you'll see that CVF files imitate the structure of a FAT volume, although it's a slightly expanded and modified one.

Clustering compressed files

The second requirement is also clearly reflected in the CVF file structure. Based on the data compression techniques used in DoubleSpace, files shouldn't be compressed in their entirety. For example, to read in a record, a DOS program opens a file and positions the file pointer on the byte with offset 35,000. Under normal circumstances this operation, which eventually must be executed by the DOS file system, doesn't present a problem. As long as you don't have a virus-infected FAT, this byte will be found just where expected - 34,999 characters after the first byte in the file.

However, since the file exists in compressed form, DoubleSpace must first locate the byte within the framework of the compressed data. Because of the compression, the byte now located at the stated offset address may actually represent byte 48,000, 120,000, or any other byte in the system. In this case, the only way to solve this problem is by completely decompressing the file, from the beginning of the file to the point at which 35,000 characters have been expanded. This process would be time-consuming. For this reason, DoubleSpace uses an alternate method of compressing files; it compresses in 8K blocks. For example, when accessing the byte with offset address 35,000 in the original (uncompressed) file, DoubleSpace can immediately find the block in which this byte is located. Although it must still decompress the entire block to reach the desired byte, all preceding blocks remain untouched.

This method of clustering of 8K blocks affects the quality of the compression negatively and positively. According to the LZW compression algorithm, references to already-encoded character strings are important. One negative aspect is the references must always refer to the current 8K block and can no longer access the file contents in their entirety. So the statistical midpoint during file compression contains fewer repeating sequences, resulting in less data being compressed through references and lowering the degree of compression. However, it would take too long to search through the entire file for the current byte to be written each time a reference is being created. Also, the entire file must be kept in memory, which also isn't feasible.

Also, too much space would be needed for the references themselves. After all, they must also contain the offset address for the character strings to which they refer. Within a closed 8K block, this offset address requires a maximum of 13 bits. Depending on its size, an entire file may require 24 bits or even 32. In the end, clustering of a compressed file into blocks of 8K each is the only practical way of ensuring the fastest and most efficient compression and decompression possible.

Now we'll return to CVF file structure. The term "cluster" is also used in connection with a FAT drive. Here the drive sectors are managed as clusters (groups) of 2, 4, 8, etc., instead of individually. In fact, the compressed 8K blocks form the basis of a DoubleSpace drive. These are the clusters that are maintained by the drive's File Allocation Table.

Differences from a normal FAT drive

In the following sections, we'll describe the various data structures within a CVF file. As you'll see, most of this information also applies to a normal DOS drive: A BIOS Parameter Block at the beginning of the drive, the boot sector, the FAT, the main directory, etc. However, two new data structures, called BitFAT and MDFAT, are also included. These data structures are needed because the compressed 8K file clusters aren't meant to be stored in identically-sized areas on the disk. Depending on the degree of compression, such a cluster will rarely require 16 sectors (8K), which is a complete cluster. This happens only if the data wasn't compressed at all or only very slightly. On average, perhaps eight 512-byte sectors will be needed (50% compression). However, in all cases the number will range from a minimum of one (maximum compression) to a maximum of 16 (no compression).

While an 8K cluster on a normal FAT drive is always given 16 sectors, therefore exactly 8K, a DoubleSpace drive must be flexible. So, instead of the usual 16 sectors, an 8K cluster is allotted only as many sectors as it still requires after compression. It is precisely in this way that sectors are saved; the additional sectors make the drive appear larger than its true size.

This is also the purpose of the two additional data structures; they reproduce, on the sector level, the cluster data from the normal FAT. So, in the granularity of memory allocation, the sector is the most important unit (the cluster affects this granularity only superficially). In a way, this undermines the idea of the cluster, although its existing structures are retained.

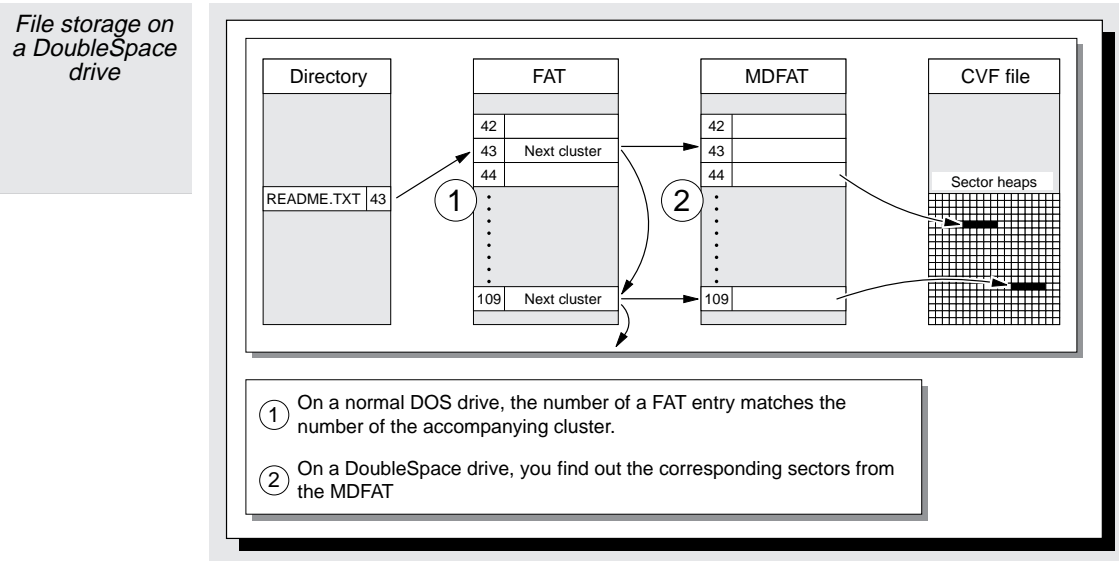
BitFAT

In this new scheme, the BitFAT acts as a type of free list, in which the still-free sectors within the CVF file are recorded. When a new 8K cluster must be written to the disk, DoubleSpace first looks here and searches for a corresponding number of free sectors. Usually it selects only adjacent sectors, so the cluster won't be scattered throughout the disk. When allocating sectors, the corresponding entries in the BitFAT are marked as occupied. When they are later freed (by deleting the file belonging to them), they are designated as free.

MDFAT

Although the FAT cannot record the sectors in which an 8K cluster is ultimately stored, the normal FAT structure had to be retained. This led to the introduction of the MDFAT. According to their original design, FAT entries point to the cluster, in which the next portion of the current file resides. This operation isn't changed by the MDFAT. However, the relationship between the number of a FAT entry and the number of the cluster, in which the data is stored, is now invalid. In a normal FAT, the sectors of the cluster belonging to FAT entry 43 are also stored in cluster 43.

So, instead of actually accessing the cluster, on a DoubleSpace drive first you must examine the 43rd entry of the MDFAT. It is only here that you can determine in which sectors cluster 43 is stored within the CVF file, and how many sectors are required for this. We'll discuss the construction of an MDFAT entry in more detail in the following sections. The following illustration shows the link between directory, FAT, MDFAT and finally, the sectors in which a compressed 8K cluster is stored:



Data structures in detail

Now that we've described the basic relationship between the various data structures in a CVF file, we'll discuss these structures in detail. You'll learn how the expanded BIOS Parameter Block and the MDFAT entries are constructed. As long as you're not planning to write programs that access these structures directly, these concepts should be easy to understand.

Data structure sequence

The following table shows the various data structures within a CVF file and their sizes. Note the sizes of the different structures and their starting points, relative to the beginning of the CVF file, depend on the size of the CVF file itself. Therefore, their positions must be determined during program runtime by reading the corresponding data from the expanded BIOS Parameter Block at the beginning of the CVF file.

All data structures, except for the sectors in the sector heap, aren't compressed. Therefore, they can be read without any special precautions.

Data structure	Sector offset	Size in sectors	Description
MDBPB	0	1	Expanded BIOS Parameter Block (Microsoft DoubleSpace BIOS Parameter Block). This is a normal BIOS Parameter Block in the format used since MS-DOS 4.0, with several additional "DoubleSpace fields". Specifically, this is where the size of the CVF file is recorded, which determines the starting points and the sizes of the data structures that follow.
BitFAT	1	*	Contains one bit for each sector in the sector heap, set to either 0 or 1 depending on whether the corresponding sector is currently being used. The size of this data structure adapts itself to that of the sector heap and thus also to the size of the CVF file. Its maximum size is 128K with a CVF file of 512 Meg.
Reserved	*	1	Free sector for use with a future version of DoubleSpace.
MDFAT	*	*	This table of 4-byte entries reproduces the FAT-entry clusters onto the sectors from the sector heap. Its size depends on the size of the CVF file, consisting of a maximum of 256K with a CVF file of 512 Meg.
Reserved	*	31	A reserved area of 31 sectors for a future version of DoubleSpace.
Boot sector	*	1	The boot sector of the CVF drive representing a copy of the host drive. It is not used for booting, but is returned upon performing a read access on Sector 0 of the DoubleSpace drive (the normal location of the boot sector).
FAT	*	*	The FAT for the DoubleSpace drive, whose structure corresponds to that of a normal DOS FAT. Its size also depends on the size of the CVF file.
Main directory	*	32	A normal DOS main directory with the usual 32-byte directory entries.
Reserved	*	2	Two additional sectors for later use by DoubleSpace.
Sector heap	*	*	The sector storage location from which are obtained the sectors for storing the compressed clusters. It fills the rest of the CVF file until the end sectors are reached.
End sector	*	>=1*	Several sectors which close the CVF file.

* = Depends on the size of the CVF file and the length of the preceding data structures.

Expanded BIOS Parameter Block (MDBPB)

With the expanded BIOS Parameter Block in MDBPB format (Microsoft DoubleSpace BIOS Parameter Block), DoubleSpace defines the structure of the CVF file. Therefore, all structures within the CVF file can be located.

The MDBPB's structure is identical up to the byte with offset 22H to the normal BPB that's been used since DOS Version 4.0. Various fields are added to this standard BPB. In DoubleSpace these fields contain information about the size and construction of the CVF file.

Using an existing CVF file as an example, you can see that usually much more space is reserved for the BitFAT, the MDFAT, and the normal FAT than is actually needed. So, enlarging the CVF file is a quick and easy process. Instead of having to move the entire sector heap to the back, which involves reshuffling almost the entire CVF file, you must make only a few changes to the appropriate MDBPB fields. In creating these data structures during setup of a DoubleSpace drive, DoubleSpace doesn't adapt itself to the CVF file size chosen by the user. Instead, it adapts itself to the size of the host drive. This is the maximum size to which the user can later expand the CVF file.

The following table shows the structure of MDBPB:

Offset	Contents	Type
+00H	Branch instruction to boot-routine	3 bytes
+03H	Manufacturer and version number	8 bytes
+0BH	Bytes per sector	1 word
+0DH	Sectors per cluster	1 byte
+0EH	Number of reserved sectors	1 word
+10H	Number of File Allocation Tables (FAT)	1 byte
+11H	Number of entries in main directory (here always 512)	1 word
+13H	Number of sectors in volume	1 word
+15H	Media descriptor	1 byte
+16H	Number of sectors per FAT	1 word
+18H	Sectors per track	1 word
+1AH	Number of read/write heads	1 word
+1CH	Distance from first sector in the volume to the first sector on the storage medium	1 word
+1EH	Total number of sectors in volume	1 dword
DoubleSpace additions:		
+22H	First sector of the MDFAT	1 word
+24H	nlog2 of the number of bytes per sector	1 byte
+25H	Number of sectors preceding the DOS boot sector	1 word
+27H	First sector of main directory	1 word
+29H	First sector of sector heap	1 word
+2BH	Number of clusters (MDFAT entries) occupied by the DOS boot sector, the reserved area and the main directory	1 word
+2DH	Number of 2K pages in the BitFat	1 byte
+2EH	Reserved	1 word
+30H	nlog2 of the number of sectors per cluster	1 byte
+31H	Reserved	5 words
+3DH	FAT type (0 = 16-bit FAT, 1 = 12-bit FAT)	1 byte
+3EH	Maximum size of CVF file in megabytes (equals the word size of the host drive)	1
+40H-1FFH	Boot routine (not needed here)	449 bytes
Length: 512 bytes		

BitFAT

DoubleSpace uses the BitFAT to keep track of the sectors from the sector heap. The BitFAT is organized as a large bit array whose bits correspond to the various sectors. A bit-value set to 1 means the corresponding sector is being used, while 0 means the sector is free. Each word from the BitFAT corresponds to 16 sectors; bit 15 represents the first sector within this group, bit 14 represents the second, etc. Unlike the other data structures, which are permanent, the BitFAT is recreated by DoubleSpace each time you start the system. The MDFAT serves as the basis for this operation.

MDFAT

The MDFAT acts as a link between the FAT and the sector heap. To access the sectors of a FAT entry, the entry corresponding to this FAT entry must first be read from the MDFAT. Each MDFAT entry encompasses 4 bytes. The following table shows its structure and the information contained in the various bit groups within an MDFAT entry:

Bits 0 - 21	Start sector	Number of the first sector from the sector heap, in which the compressed cluster is stored. Successive sectors contain the remaining bytes of the compressed cluster.
Bits 22 - 25	Size of compressed cluster	Provides the number of sectors that were required for compressed storage of the cluster. The minimum is one sector while the maximum is 16. The values 1 to 16 are represented by 0 to 15, so the number 0 represents one sector.
Bits 26 - 29	Size of uncompressed cluster	Records the uncompressed size of the cluster, which is normally 16. The only exception is the last cluster of a file when the file size is not a multiple of the cluster size of 16 sectors. Here also the values from 1 to 16 are represented by the numbers 0 to 15. So, this field usually contains the number 15, representing 16 sectors.
Bit 30	0=Sector compressed 1=Sector uncompressed	This flag shows whether the contents of the cluster were stored in compressed or uncompressed form. If, when compressing clusters, DoubleSpace doesn't save at least one sector, it forgoes the compression and stores the cluster as uncompressed.
Bit 31	0=MDFAT entry unused 1=MDFAT entry used	Shows whether the MDFAT entry is currently being used. This bit is maintained for DOS Undelete programs, which, when restoring files, mark the FAT clusters as again being used.
Total 32 bits		

Boot sector

The boot sector contains a 1:1 copy of the boot sector but doesn't actually boot the DoubleSpace drive. It's stored here only for compatibility reasons.

FAT

The File Allocation Table of a DoubleSpace drive acts just like a normal FAT that you would find on an uncompressed drive. However, unlike a normal drive, DoubleSpace maintains one FAT instead of two FATs. During read/write accesses by DOS programs, the second FAT is virtualized, so these programs don't notice the missing second FAT. Therefore, read accesses to the second FAT are rerouted to the first one while write accesses to the second FAT are ignored. Only write accesses to the first FAT are reflected in the FAT of a DoubleSpace drive.

Main directory

The structure of the main directory of a DoubleSpace drive is identical to that of an uncompressed drive. Here also the files and subdirectories stored in the main directory are represented by 32-byte entries. The main directory of a DoubleSpace drive contains space for 512 of these entries.

Sector heap

The sectors in the sector heap store the compressed files and subdirectories, which in this context are treated just like files. Sectors that contain compressed data always begin with a four-byte "tag", which describes how the data were compressed. For the standard DoubleSpace algorithm, the tag consists of the following bytes (uncompressed sectors have no tag):

```
4DH 44H 00H 00H
```

DoubleSpace and the boot process

So the user can address the compressed DoubleSpace drive in the same way as the original host drive, DoubleSpace is automatically started during the boot process. This occurs after loading and starting IO.SYS, which is one of the two core modules of DOS. At this point, all hardware tests have been executed, the active partition selected, and the boot sector contained therein loaded and started. In other words, DOS is on its way to taking control of the system.

Once IO.SYS is started, it begins by searching for the file DBLSPACE.BIN, which contains the program code for accessing compressed drives. If it finds the file in the main directory of the drive, it loads and executes the file. However, if the file is missing, IO.SYS follows its usual path of initializing the file system, completely oblivious to the possible existence of a DoubleSpace CVF file.

Once started, DBLSPACE.BIN first opens the initialization file DBLSPACE.INI, which lists the names of the CVF files on the drive and their future drive designations. DBLSPACE.BIN now mounts these CVF files (incorporates them as normal drives) into the DOS system. If DBLSPACE.BIN finds a CVF file with the number 000 (DBLSPACE.000), then this file's device ID is switched with that of the host drive, since DBLSPACE.000 always refers to the compressed contents of the host drive. The mounted drives from this moment on can be addressed, just like all other DOS drives, through their device IDs.

If DBLSPACE.BIN doesn't find a CVF file, it removes itself from memory and returns control to IO.SYS. This mechanism was included so you could also record on diskettes with the DBLSPACE.BIN file, without them actually being compressed. After successfully mounting drives, the system continues as usual, processing the lines in the CONFIG.SYS file. Usually the kernel will arrive at the following line:

```
devicehigh=c:\dos\dblspace.sys /move
```

DoubleSpace inserts this line into CONFIG.SYS as part of its setup routine. Its function is to load DoubleSpace into upper memory (if available) so it no longer uses memory under 640K. The driver DBLSPACE.SYS's only purpose is to move the DoubleSpace kernel from DBLSPACE.BIN into Upper Memory. So, by itself DBLSPACE.SYS isn't involved in disk access using DoubleSpace.

This second "initialization" of DoubleSpace is necessary because, when DBLSPACE.BIN executes, no upper memory exists. Upper memory only appears after HIMEM.SYS (a file called from CONFIG.SYS) accesses upper memory. Since DoubleSpace anticipates this shift, DBLSPACE.BIN is placed at the top of low memory, immediately below the 640K limit. This ensures that, when it's later removed, a gap isn't left in lower memory and valuable memory wasted. If a call for DBLSPACE.SYS is included within the CONFIG.SYS file, upon completing the processing of this file, DOS automatically takes over the task of shifting the program code from DBLSPACE.BIN into lower RAM (instead of into upper memory).

DoubleSpace and applications

In relation to DOS commands, DoubleSpace is completely transparent to the user. Similar to other storage devices, a DoubleSpace drive can be addressed via its respective drive designation. DoubleSpace is equally transparent at the level of application programs, although you must consider different types of disk access. At the topmost level, application programs access files and directories using DOS interrupt 21H functions. At some point during the "processing chain" that initiates the call to these functions, DoubleSpace also becomes part of the file system. Using the mechanism described earlier, it finds its way to the appropriate sector from the CVF file sector heap, in which the desired bytes are stored or should be stored. The data are then either compressed or decompressed, depending on the type of access.

The next level involves DOS interrupts 25H and 26H, with whose help DOS programs can directly read and write to individual sectors of a volume. While most application programs avoid these functions, they're basic for many DOS utilities and how they function. This especially applies to programs such as Norton's DirectorySort and defragmentation programs.

However, even to these programs, DoubleSpace provides the illusion of a normal DOS drive because it intercepts the two interrupts and converts the given sector numbers (via the MDFAT) to the sectors in which the respective information is actually stored or should be stored. Again, this also includes compression or decompression. The imitation goes so far that even DOS 5.0 defragmentation programs can be run under DOS 6 in combination with DoubleSpace. This occurs despite the fact that defragmentation is considered a very delicate operation.

In any case, there is not much to be gained from having such programs on a DoubleSpace drive, since they only recombine the sectors of the file as they are defined in the FAT. They neglect to rearrange the various compressed cluster sectors in relation to their order in the sector heap. Therefore, on DoubleSpace drives we recommend installing the DEFRAG program included with DOS 6.2. This program takes the sector heap into consideration and restores order here as well.

DoubleSpace software interface

From the outside you can address DoubleSpace through a software interface that depends mainly on an expansion of multiplexer interrupt 2FH. The eight functions address the requirements of disk utilities exclusively and aren't intended for normal application programs. After all, DoubleSpace remains completely transparent to DOS programs if they use the usual methods for accessing files, drives, and directories. This table summarizes the eight multiplexer functions of DoubleSpace.

Function	Description
00H	Obtain version information
01H	Scan drive map
02H	Switch drive ID
05H	Link compressed drive
06H	Deactivate a DoubleSpace drive
07H	Establish storage space
08H	Obtain information about CVF file fragmentation
09H	Scan number of compressed drives

All these functions must be called with the value 4A11H in the AX register and the function number in the BX Register. The code 4A11H acts as an identifier for DoubleSpace within the framework of the multiplexer interrupt. Also, all functions in the DL register await the device ID of the DoubleSpace drive currently to be addressed, where 0 stands for A:, 1 for B:, etc.

Function 0000H checks for the existence of DoubleSpace and obtains information about the drive designations used by DoubleSpace. This function call should precede all others, to ensure that DoubleSpace is resident in memory. It will also indicate whether DoubleSpace is in upper memory, or in conventional memory under 640K.

Function 0001H determines whether a particular drive is compressed and whether its drive identifier is genuine (i.e., not switched with the host drive). Function 0002H performs this drive identifier exchange, while function 0005H links drives. "Normal" DoubleSpace drives are linked automatically at system start, so this function is primarily designed for drivers governing interchangeable media. You can exchange any storage medium compressed by DoubleSpace and make it available to DoubleSpace in your system. In the opposite direction, Function 0006H provides a way for these programs to free a linked drive from DoubleSpace control, in which case it is no longer accessible under its previous device ID. It is not deleted however; its CVF file remains exactly as is.

Function 0007H determines the size of the sector heap within the CVF file and the maximum amount of compressed data. It doesn't indicate the maximum amount of data in uncompressed form since this depends on the degree of compression that you can't predict for data yet to be stored. This function also provides the number of sectors inside the sector heap. Function 0008H offers some additional information of rather doubtful value, referring to CVF file fragmentation on the host drive. This function returns the same value as the MaxFileFragments setting in the DBLSPACE.INI configuration file. Similarly, Function 0009H is closely connected with another setting from this file. It returns the MaxRemovableDrives parameter setting. When DoubleSpace boots up, this function decides how many of a certain structure named DISK_UNIT DoubleSpace will create. One of these structures occupies 96 bytes and is required for the management of each active DoubleSpace drive.

Besides the multiplexer functions, DoubleSpace offers two other functions, accessible from subfunction 04H of the DOS IOCTL function, rather than the multiplexer interrupt. The reason why these tasks are split up is simple. When you call functions through the multiplexer interrupt, DOS doesn't set the InDos flag, which enables reiterative calls to the function. In other words, while a multiplexer function is running, it can be called a second time by a TSR program or by another virtual DOS machine under Windows. With the eight multiplexer functions, this potential for reentry doesn't create a problem. However, this doesn't apply to the two IOCTL functions, since these relate to the internal caches that DoubleSpace maintains for temporary storage of cluster data and sectors from the MDFAT and BitFAT. Caches are susceptible to reentry problems.

In concrete terms, these two functions let you store the contents of the DoubleSpace caches onto disk, which is important if you want to exchange media or reset the system. For example, when a new drive is first installed, DoubleSpace can no longer write the contents of these caches onto the old one because data on the old drive can get lost. What's worse is that now the cache contents may be written to the new disk, which would guarantee data losses from the new drive.

Therefore, any action, such as exchanging compressed media, must always be preceded by a call to one of the DoubleSpace IOCTL functions. The difference between them lies simply in the second simultaneously declares the cache contents invalid, while the first keeps them valid.

A detailed listing of all DoubleSpace multiplexer functions can be found in Appendix E on the companion CD-ROM. The two IOCTL functions are listed with the DOS API functions in Appendix D.

MRCI compression interface

DoubleSpace interacts closely with a software interface called MRCI (Microsoft Real-time Compression Interface). MRCI gives application programs, TSR programs and device drivers access to an MRCI server, which compresses and decompresses data blocks. An MRCI server loads into memory as a component of DoubleSpace during bootup. Although DoubleSpace itself uses MRCI for compressing and decompressing data blocks, other programs can also access it. For example, the DOS 6 backup program uses the MRCI server to store backup data in compressed form. It is also used by the Flash File System. The Flash File System from Microsoft is used in combination with flash-memory cards conforming to the PCMCIA standard, which serve as substitutes for hard disks.

Microsoft Real-time Compression Format

The data here are compressed according to a fixed format, called Microsoft Real-time Compression Format, or MRCF. Following this format guarantees that data compressed on one MRCI server can be decompressed on another MRCI server. The MRCF format results in a "loss-free" compression, which means there are no differences between the compressed data and the subsequently decompressed data. Although this seems only natural, it's precisely what distinguishes this type of compression from other processes, such as JPEG or MPEG, which "calculate away" data to a certain extent, to achieve higher compression rates. Such a thing never occurs with MRCI servers.

Hardware servers

MRCI servers can be implemented in hardware as well as software. Until now the DoubleSpace MRCI server has existed only as a software implementation. However, there is nothing to stand in the way of third-party manufacturers also offering a hardware version. The biggest advantage of a hardware implementation would be an increase in speed, since dedicated compression/decompression hardware could accomplish this task much faster than the CPU running the corresponding software. Hardware enhancements of this type are assigned to a fast bus so they can receive and transmit the data as quickly as possible. Therefore, the first hardware servers will probably be designed for the VL bus or the PCI bus.

MRCI clients

While Microsoft will reveal the secrets of its MRCI server only to selected hardware manufacturers, the development of MRCI clients is open to all. MRCI clients are application programs, TSRs, or device drivers that use MRCI servers to compress or decompress data. The MRCI server can be used equally well by a terminal program compressing files to be transmitted, a program transmitting images over a network to various workstations, or a program compressing data for backup.

An MRCI server makes five different tasks available to an MRCI client, as shown in the table on the right. The names listed here have only symbolic meanings, since access to the MRCI server occurs through a single point of entry, instead of through specialized functions. For this point of entry, a constant must be passed to register AX to access the desired function.

Establishing contact by MRCI client

An MRCI client must call MRCQuery first, since it is only through this function that it gains entry into the MRCI server, through which it can then call the other functions. MRCQuery is called using multiplexer interrupt 2FH, whereby 4A12H must be placed in register AX as an MUX code for the MRCI server. To prove that it is a legitimate MRCI client, the ASCII codes for the character combination "MR" must be placed in register CX, and those codes for "CI" must be placed in register DX. The server transposes these character combinations so register CX receives "IC" and register DX receives "RM."

Although at first this procedure may seem rather strange, it guarantees the MRCI client will be certain that it is dealing with an MRCI server. Any other program can engage the multiplexer interrupt under the code number 4A12H, yet none will answer the function call with precisely the same inversion of registers CH, CL, DH, and DL. If this exchange didn't occur, then an MRCI server hasn't engaged the multiplexer interrupt. This means there is no MRCI software server in the system. However, this doesn't mean that a hardware server doesn't exist. This server enters the system through BIOS interrupt 1AH by using

function B001H. Function B001H, newly developed by Microsoft, serves the same purpose as the multiplexer interrupt call, applying to registers CX and DX.

If this call is also unsuccessful, then an MRCI server doesn't exist in the system. However, if one of these two calls returns the desired combination in the register pair CX/DX, then the register pair ES:DI will contain the address of what is known as an MRCI info structure (see following table), which contains information about the MRCI server, its capabilities, and most importantly, the point of entry for calls to the various MRCI functions.

Name	Purpose
MRCQuery	Obtains information about an installed MRCI server
MRCCompress	Compresses data block with standard compression
MRCDecompress	Decompresses data block
MRCMaxCompress	Compresses data block to maximum
MRCIncrementalDecompress	Incrementally decompresses data block

Offset	Meaning	Type
+00H	4-byte ASCII code with manufacturer name ("MSFT" for Microsoft)	4 bytes
+04H	Version number of MRCI server High byte contains main version number, low byte the sub-version number	1 word
+06H	MRCI version upon which server is based High byte contains main version number, low byte the sub-version number	1 word
+08H	MRCI server entry point for calling MRCI functions	1 var ptr
+0CH	Flag for server capabilities (see below)	1 word
+0EH	Hardware flag for server capabilities implemented through hardware	1 word
+10H	Maximum data block size that server can compress	1 word
Length: 18 bytes		

The two flags returned within the structure are identical in their makeup. The first flag indicates which functions are available overall, while the second flag indicates which of these are implemented through hardware. If a bit is set in the first flag and not in the hardware flag, then the corresponding function is available through the software and not the hardware.

While all MRCI servers support standard compression and decompression, this isn't true for maximum compression and incremental decompression. Maximum compression involves reducing the data even further than is possible with standard compression. However, doing this increases compression time. With incremental decompression, the idea is to decompress a compressed data block only up to a certain byte. This is useful when you need only a certain number of bytes instead of the entire data block. Instead of wasting time decompressing the entire block, with incremental decompression you can stop the unpacking of data at a certain byte and continue from that point later as desired. To determine whether each of these two functions is available, check the corresponding flags within the MRCI info structure. One entry which you usually don't have to worry about is the maximum size of compressible data blocks. This is always at least 8K for MRCI servers. So checking the corresponding element within the data structure is unnecessary for block sizes up to 8K.

Calling the MRCI server functions

In the previous table, the various bits for the individual flags are purposely listed along with their order. Upon calling the MRCI server, these orders are the ones that must be given as a function code in register AX, as the entry point to the MRCI server. The MRCI server also looks in register CX for the value 0 or 1, depending on whether the client is a transient application program (0) or a resident system component (1), such as a TSR or device driver.

Also, upon calling the MRCI server it awaits two FAR pointers in the register pairs ES:BX and DS:SI. ES:BX is for the pointer to the MRCI info block returned from the Query call, and DS:SI is for a pointer to an MRCI request block. Here the MRCI server obtains important information necessary for running the corresponding function (e.g., buffer addresses), where files to be compressed or decompressed are located. The table on the following page shows the precise layout of this structure.

The first four fields of the data structure describe the locations of the source and destination buffers in memory, and their length. During compression, data from the source buffer are compressed into the destination buffer (i.e., following a

Offset	Contents	Type
+00H	FAR pointer to source buffer	1 var ptr
+04H	Length of source buffer in bytes	1 word
+06H	Reserved	1 word
+08H	FAR pointer to destination buffer	1 var ptr
+0CH	Length of destination buffer in bytes	1 word
+0EH	Block size for compressed data	1 word
+10H	Pointer for incremental decompression	1 var ptr
Length: 18 bytes		

successful call to the function, the destination buffer contains a compressed version of the uncompressed data from the source buffer). The same occurs with decompression whereby the contents of the source buffer are decompressed into the destination buffer. The MRCI server requires that both buffers stay within their limits and, therefore, considers the given buffer sizes.

For the source buffer, the number of bytes to be compressed or decompressed is given during compression/decompression. The MRCI server uses this information to determine how much space is available in the destination buffer for the compressed or decompressed data. If there

isn't enough space, the server function returns to the client with an error code. Therefore, there is no danger of overwriting the destination buffer. From the length of the destination buffer, you can also determine, following compression or decompression, the number of bytes in the compressed/decompressed data.

Offset 0EH (Block size for compressed data) applies only to data compression. This offset is designed to speed up the functioning of the MRCI server. Usually compressed data is stored in blocks of constant size. In DoubleSpace this is on the lowest level of the hard drive sector, which always contains 512 bytes. When compressing data, it's useless to carry the compression beyond 512 bytes, since you would still need the entire sector anyway. Even if you managed to squeeze the data into 300 or 200 bytes, as far as hard drive space is concerned it would make no difference. Therefore, the most practical method is stopping the compression at the 512-byte limit and saving the time that would otherwise have been used in continuing it. Valid entries in this field are 1 to 32768. DoubleSpace sets the value to 512, but application programs can set it to 1 if they need a high degree of compression.

Server calls and Windows

The instruction code in AX, the type of application in CX, a pointer to the MRCI info block in ES:BX, and a pointer to the MRCI request block in DS:SI are needed to call an MRCI server function. However, first the program must enter a Windows Critical Section, for the case when the application is running in a virtual DOS machine under Windows in 386 Enhanced mode. Since true multitasking is being performed among these VMs, the MRCI server may be confronted with several function calls from different VMs simultaneously. The MRCI server can't handle this situation.

This problem is avoided by preceding the server call with entry into a Windows Critical Section. In a Critical Section of this type, only one VM or Windows application can exist at any given time. If another VM has already claimed this attribute, the call for entry into the Critical Section is held up until the other VM has left its Critical Section. The call for exiting a Critical Section is as important as the call for entering a Critical Section. The entry code for a Critical Section and the subsequent exit code is precisely stipulated by Windows, and must be implemented in assembly language. The entry code is as follows:

```

Entry code  Exit code
push        ax      push        ax
mov         ax,8001h  mov         ax,8101h
int         2ah      int         2ah
pop         ax      pop         ax

```

If Windows isn't active these calls go into oblivion, since no special handler exists for these calls after interrupt 2AH. The program will continue execution, instead of crashing the system. Although there is no entry into a Critical Section, once Windows is inactive this doesn't matter anyway. Before calling a server function, TSR programs and device drivers must take control of the InDos flag. This is necessary to prevent a reentrance of DOS (see Chapter 35 for information on TSR programming). Following a call to the MRCI server, a status value is returned in register AX. A value of 0 means the function was executed successfully, while all other values represent errors in accordance with the table on the right.

Code	Error
0	All OK
1	Incorrect function code
2	Server is busy
3	Destination buffer too small
4	Data cannot be compressed

34

Terminate And Stay Resident (TSR) Programs

Since it was introduced, DOS has been criticized for its inability to handle multitasking (running more than one program simultaneously). Although OS/2 is capable of multitasking, it requires an AT or 80386 (or higher) computer to run. But TSR (Terminate and Stay Resident) programs can provide DOS machines with some of the advantages of multitasking. This type of program moves into the "background" once it's started, and becomes active when the user presses a particular key combination. The SideKick program produced by Borland International made TSR programs very popular.

Running a TSR program isn't true multitasking because only one program is actually running at any given time. However, by pressing a key, the user can immediately access useful tools, such as a calculator, calendar, or note pad. In addition to these applications, macro generators, screen layout utilities, and text editors are also available in TSR form. Many TSR programs can even interact with the programs they interrupt and transfer data between the TSR and the interrupted program. An example is a TSR appointment book that inserts a page from its calendar in a file that's loaded into a currently running word processor. Although many different applications can be implemented with TSR programs, these programs share two characteristics:

- They operate in basically the same way
- They are based on similar programming concepts

In this chapter, we'll examine these two items and present simple implementations of TSR programs. First, remember this involves very complex programming. So, to understand this material, you must know how things operate within the system. This especially applies to TSR programs, because, by their definition, they practically ignore the single-task nature of DOS, in which one program has access to all the system resources (RAM, screen, disk, etc.).

A TSR program must contend with many other elements of the system, such as the BIOS, DOS, the interrupted program, and even other TSR programs. Managing this is a difficult task, and can only be accomplished using assembly language. Of the available PC languages, only assembly language offers the ability to work at the lowest system level, the interrupt level. However, although it has this capability, assembly language is as flexible as high level languages for writing TSR applications, such as calculators or note pads. Because of this, we'll list two assembly language programs in this chapter. These programs will allow you to "convert" Turbo Pascal, Turbo C, and Microsoft C programs into TSR programs.

Activating A TSR Program

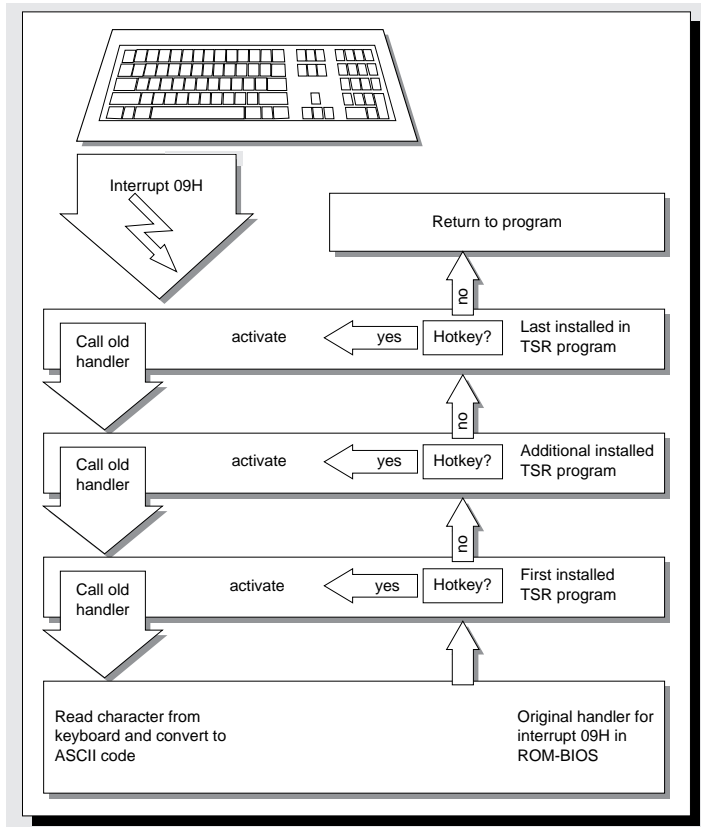
Let's begin by discussing how a TSR program is activated. To place our TSR program in the foreground immediately after we press a certain key combination (called the hotkey), we must install some sort of activation mechanism that's tied to the keyboard. We can use interrupts 09H and 16H, which are two system keyboard calls. Interrupt 16H is the BIOS keyboard interrupt, which programs use to read characters and keyboard status. If we use this interrupt, then our TSR program can only be activated when the main program is using interrupt 16H for keyboard input.

So, instead we should use interrupt 09H, which is called by the processor whenever a key is pressed or released. We can redirect this interrupt to our own routine, which can check to see whether the TSR program should be activated. Before it does this, the routine should call the old interrupt 09H handler. There are two reasons for this. The first reason is related to the task of interrupt 09H, which informs the system the keyboard needs the system's attention to transfer information about a key event. So, interrupt 09H usually points to a routine, within the ROM BIOS, that accepts and evaluates information from the keyboard. Specifically, it receives the code from the keyboard, converts it to an ASCII code, and then places this code in the BIOS's keyboard buffer. Since our TSR program neither wants nor is able to handle this job, we must call the original routine; otherwise keyboard input will be impossible.

The second reason is that other TSR programs may have been installed first. These routines would have redirected interrupt 09H to their own routines. Since our program is in front of these programs in the interrupt handler chain, their interrupt routines won't be called automatically if we don't call the old interrupt handler. So, we wouldn't be able to activate these TSR programs. Remember that when a TSR program is called using a redirected interrupt routine, it should always call the old interrupt handler before or after its own interrupt processing.

The call cannot be made with the INT assembly language instruction, because this would simply recall our own interrupt handler. Usually this leads to an infinite loop, a stack overflow, and, eventually, a system crash. To avoid this, we must save the address of the old interrupt handler when the TSR program is installed. We can then call the old interrupt handler with this stored address by using a FAR CALL instruction. To simulate calling this handler through the INT instruction, we must first place the contents of the flag register on the stack with the PUSHF instruction before the CALL.

*Reading keys
for TSR
programs using
interrupt 09H*



Following the interrupt handler call, the contents of port 60H are read. This port contains the scan code of the key most recently pressed (see Chapter 9 for more information on scan codes). The previous interrupt handler, which is called later, also reads the contents of this port to determine the scan code. When the new interrupt handler reads this port, the port's content remains unchanged, so when the old handler reads its contents later, the same value will be returned. This value remains constant until a new key is pressed.

From the contents of keyboard port 60H, the new TSR interrupt handler can determine whether the user has pressed its designated hotkey. However, this value will correspond to only part of the hotkey, since hotkeys are generally used as key combinations, often combining a letter or number with one or more modifier keys (**Shift**, **Ctrl**, **Alt**, etc.).

After the return from the interrupt handler, we can check to see whether the hotkey was pressed to activate the TSR. The BIOS keyboard status byte (BIOS variable address 0040:0017) indicates the status of the following keys:

- | | |
|--------------------------|---|
| ➤ Right Shift key | ➤ Left Shift key |
| ➤ Ctrl key | ➤ Alt key |
| ➤ Num Lock key | ➤ Scroll Lock key |
| ➤ Caps Lock key | ➤ Sys Req key (AT keyboard only) |

If the appropriate keys are pressed, the user is trying to activate the TSR program. This is only possible if certain conditions are met, all of which are based on the fact the DOS is not reentrant.

DOS

Since the TSR program can be activated from the keyboard at any time, regardless of the other processes in the system, it may interrupt a call to a DOS function. This may not lead to problems as long as the TSR program properly returns to the interrupted DOS function. However, a problem occurs when the TSR itself tries to call DOS functions, which is difficult to avoid when programming in a high level language. This demonstrates the problem of reentry. This refers to the ability of a system to allow multiple programs to call and execute its code at the same time. However, DOS is not reentrant because it is a single-task system and assumes that DOS functions will be called in sequence, instead of in parallel.

Calling a DOS function from within a TSR program while another function is executing, leads to problems because the processor register SS:SP is loaded with the address of one of three DOS stacks when interrupt 21H is called. Which of the three stacks is used depends on the function group, to which the DOS function belongs, and cannot be determined by the caller. While the DOS function is being executed, it places temporary data, as well as the return address to the calling program, on this stack. If the execution of the function is then interrupted by the activation of a TSR program, which then calls a DOS function, DOS will again load register pair SS:SP with the starting address of an internal stack. If it's the same stack the interrupt function was using, each access to the stack will destroy the data of the other function call. The DOS function called by the TSR program will be executed properly, but the problem will occur when the TSR program ends and control returns to the interrupted DOS function. Since the contents of the stack have been changed in the meantime by other DOS calls, the DOS function will probably crash the system.

Bypassing reentry

Two ways are available to avoid these reentry problems. Either avoid calling DOS functions or allow the TSR program to be activated only if DOS functions aren't being executed. Since we've already ruled out the first option, we must use the second. DOS helps us by providing the INDOS flag, which is normally used only inside DOS but which is very useful to us as well. This flag is a counter that counts the nesting depth of DOS calls. If it contains the value 0, no DOS functions are currently being executed. The value 1 indicates the current execution of a DOS function. Under certain conditions, this counter can also contain larger values, such as when one DOS function calls another DOS function, which is allowed only in special cases.

Since there isn't a DOS function that can read the value of this flag, we must read the contents directly from memory. Since the address doesn't change after the system is booted, we can obtain the address when the TSR is installed and save it in a variable. DOS function 34H returns the address of the INDOS flag in register pair ES:BX.

This flag is read in the interrupt handler for interrupt 09H since it checks to see whether the hotkey was pressed. It also allows the TSR program to be activated only if the INDOS flag contains the value 0. However, this doesn't completely solve our problem. It coordinates the activation of the TSR program with DOS function calls of the transient program being executed in the foreground, but it doesn't allow the TSR program to be called from the DOS user interface. Since the DOS command processor (COMMAND.COM) uses some DOS functions for printing the prompt and accepting input from the user, the INDOS flag always contains the value 1. In this case, we can interrupt the executing DOS function but need to ensure the INDOS flag contains the value 1. A DOS function can be called from transient program or from the DOS command processor.

However, there is also a solution to this problem. It involves the fact the DOS is in a kind of a wait state when it's waiting for input from the user in the command processor. To avoid wasting any valuable processor time, it periodically calls interrupt 28H, which is responsible for short term activation of background processes, such as the print spooler (DOS PRINT command) and other tasks. If this interrupt is called, it's relatively safe to interrupt DOS and call the TSR program.

To use this procedure, a new handler for interrupt 28H is installed when the TSR program is installed. First it calls the old handler for this interrupt and then checks to see whether the hotkey has been pressed. If this has occurred, the TSR program can be activated, even if the INDOS flag isn't 0. We still must add another restriction. The TSR program cannot even be activated with the handler for interrupt 09H if time-critical actions are being performed in the system.

Time-critical actions

These are actions which, for various reasons, cannot be interrupted because they must complete execution in a relatively short time. In the PC, this includes accesses to the floppy and hard disk, which at the lowest levels are controlled by BIOS interrupt 13H. If an access to these devices isn't completed by a certain time, the system can be seriously disrupted. A dramatic example of this is if the TSR program performs an access to these devices before another access, which is initiated by the interrupted program, has finished. Even if this doesn't crash the system, it will lead to data loss.

Avoid this by installing a new interrupt handler for BIOS interrupt 13H. When this handler is called, it sets an internal flag that shows the BIOS disk interrupt is currently active. Then it calls the old interrupt handler, which performs the access to the floppy or hard disk. When it returns to the TSR handler, the flag is cleared, signaling the end of BIOS disk activity. To prevent this interrupt handler from being interrupted, the other TSR interrupt handlers monitor this flag and will activate the TSR program only if the flag indicates the BIOS disk interrupt isn't active.

Delayed activation

Depending on the current DOS or BIOS operations, occasionally a TSR program may be unable to move into the foreground. Because of this, most TSR programs also install an interrupt handler for timer interrupt 08H. This interrupt handler can delay the start of the TSR. If the hotkey is recognized and the TSR program is unable to execute at that moment, a special flag is set within the keyboard interrupt handler.

This flag is then checked by the new timer interrupt handler, which is called 18.2 times a second if the current application hasn't changed the timer frequency. If the handler discovers the TSR program is waiting to be activated, and if DOS and BIOS have completed their operations, then the TSR program can be activated. However, a time limit should be set for this delay. Otherwise, if the hotkey is pressed, and if DOS is currently executing a lengthy operation, the TSR program may only be activated after several seconds. If the length of this delay isn't limited, the user won't know whether the TSR program hasn't recognize the hotkey, or whether it's still waiting for an opportunity to start.

So, the flag that facilitates the delayed start of a TSR program also acts as a timekeeper. Its value decrements each time the timer interrupt is activated. When this value reaches 0, the interrupt handler stops trying to activate the TSR program. If the keyboard interrupt handler initializes this flag with a value of 6, for instance, then the maximum delay for the start of the TSR program consists of 6 timer interrupts, or about one third of a second. If the program cannot be activated within this time frame, the hotkey call has no effect.

Recursion

Since the hotkey can still be pressed after the TSR program has been activated, we must prevent the TSR program from being reactivated before it's finished. We can simply add another flag, which is checked before the TSR is activated. The TSR program sets this flag when it begins and clears it again just before it ends. If an interrupt handler determines that this flag is set, it will simply ignore the hotkey. Once all these conditions have been met, we can activate the TSR program.

Context switch

The process of activating a TSR program is called a *context switch*. The program context or environment is the only information needed for operating the program. This includes such things as the contents of the processor registers, important operating system information, and the memory occupied by the program. We don't have to worry about the program memory in our context switch, however, since our TSR program is already marked as resident, which means the operating system won't give the memory it occupies to other programs.

The processor registers, especially the segment registers, must be loaded with the values the TSR program expects. These are saved in internal variables when the TSR program is installed. Since the contents of these and other registers will be

changed by the TSR program, the contents of the registers must be saved because they belong to the context of the interrupted program and must be restored when it starts again.

This also applies to context dependent operating system information, which, for DOS, includes only the PSP (Program Segment Prefix) of the program and the DTA (Disk Transfer Area). The addresses of both structures must be determined and saved when the TSR program is installed so they can be reset when context is changed to the TSR program. Also, remember to save the addresses of the PSP and DTA of the interrupted program before the context change to the TSR program. There are DOS functions for setting and reading the address of the DTA (DOS functions 1AH and 2FH), but there are no corresponding documented functions for the PSP. DOS Version 3.0 includes function 62H, which returns the address of the current PSP, but has no function for setting the address. Undocumented functions for doing both exist in DOS 2.0: function 50H (set PSP address) and 51H (get PSP address). Both of these are used in our TSR demonstration program.

The TSR code must perform one final task. When the TSR program is activated using interrupt 28H, an active DOS function is interrupted. This function's stack shouldn't be disturbed. Generally we should take the top 64 words from the current stack and place them on the stack of the TSR program. This completes the context change to the TSR program, which means the TSR program can now be started. At the moment, the TSR program can be viewed as a completely normal program which can call arbitrary DOS and BIOS functions. The only competitor left in the system is the foreground program. The TSR must ensure that it leaves both the foreground program and its screen undisturbed.

Saving the screen context

The tasks were exclusively handled in assembly language. However, the C or Pascal program comprising the TSR program itself can save the screen context. This screen context includes the current video mode, the cursor position, and the screen's contents. The contents of the color registers and other registers on the video card must also be saved, if any of these values are changed by the TSR program. The video mode can easily be determined with function 00H of BIOS video interrupt 16H (see Chapter 4). If the screen is in text mode (modes 0, 1, 2, 3, and 7), the TSR program must save the first 4000 bytes of video RAM. The video BIOS can be used for this or you can access the video RAM directly (see Chapter 4).

Saving the video mode becomes very complicated if a graphics mode is active, since the video RAM for EGA and VGA cards can be as large as 256K in some modes. If the TSR program interrupted a transient program, it may be impossible to allocate a large enough buffer to handle both programs. This is why many TSR programs won't activate themselves from within graphics mode, and can only be used in text mode. Since PCs mostly use text mode, this isn't a major problem. Microsoft Windows_{3.1}, which operates only in graphics mode, is an exception. Since this program usually supports some mechanism for parallel execution of calculators, note pads, etc., TSR programs aren't very useful under Windows.

TSR Programs In Pascal And C

You must understand the information we just presented to completely comprehend the assembler modules TSRPA.PAS (for Pascal programs) and TSRCA.ASM (for C programs). Since both modules are based on the same basic principles and the differences between these programs are limited to the different conventions found in Pascal and C, we'll discuss only their structure.

Both assembly modules can install the TSR program upon the first program call from the DOS command line, and can reinstall the program upon a second call. They also offer a TSR program, being called from the DOS command line, the option of communicating with a memory-resident copy of the same program. This makes it possible, for example, to specify a new hotkey for the installed version, without having to remove the program from memory and then reinstall it. Other parameters can also be changed in this way, since any desired Pascal or C routine can be called within the memory-resident TSR program, as you'll see below.

Assembly language modules

To support the mentioned functions, the assembly module offers the high level language program seven procedures, which are listed in the table below.

Name	Description
TsrInit	Transforms the program into a TSR program, installs the interrupt handler, ends the program and installs it in memory.
TsrIsInst	Determines whether a copy of the program is already resident in memory.
TsrCanUnins	Determines whether the resident copy of the program may be uninstalled.
TsrUnInst	Removes a memory-resident copy of the program from memory.
TsrSetPtr	Sets a pointer to the address of the procedure that is to be called within the memory-resident copy of the program.
TsrCall	Calls the procedure identified by TsrSetPtr.
TsrSetHotkey	Sets the program's hotkey.

Checking the installation status

The high level language program must first call the TsrIsInst procedure to determine whether a copy of the program is already installed in memory as a TSR program. To do this, the procedure uses the DOS multiplex interrupt 2FH (MUX). This is possible because an interrupt handler for interrupt 2FH is also installed when a TSR program is loaded into memory. This interrupt only responds to a very specific function, whose function number is determined when TsrIsInst is called. If it's called with another function number, the TSR program simply passes the call to the previous interrupt handler. The new MUX interrupt handler supports two functions as subfunctions of the specified function number, with the subfunction numbers AAh and BBh. The first subfunction is used to locate a resident version of the TSR program.

As is commonly the case with MUX functions, the function and subfunction numbers in registers AH and AL are simply swapped when the subfunction is called. However, if the TSR program isn't installed, this swap doesn't occur, because none of the previous MUX handlers recognize the function. The contents of the AX register are therefore returned unchanged.

In calling this MUX function, TsrIsInst can easily determine whether the program has already been installed. If this is the case, TsrIsInst also calls the second TSR program's second MUX function, which returns the resident program's segment address. This value is then stored in a variable within the TSR program. Like all other variables of the assembly module, this variable is stored in the module's code segment. This ensures the variables can also be addressed within the interrupt handler, even if the program's data segment cannot be accessed.

You now have enough information to understand how the two assembly language interfaces operate. The two programs are based on the principles we've outlined here; the differences between them reflect the different syntaxes of compiled C and Pascal programs. First, we'll concentrate on the similarities between the two programs. Both programs assume the TSR program was installed by the first call from the DOS level and will be reinstalled on each new call. It's important to remember one general rule: a TSR program can be reinstalled only if no other TSR programs have been installed in the meantime. The LIFO (Last In, First Out) principle applies here, so the only way a TSR program can be reinstalled is if it was the last one to be installed, and if the corresponding interrupt vectors point to its interrupt handlers. If another TSR program was installed after it, the interrupt vectors point to its handlers.

To support this mechanism, the assembly language interface offers the high-level program three routines to install and later reinstall the TSR program. To decide whether the program should be installed or reinstalled, the first function should be called to determine whether the TSR program is already installed. This routine is passed an identification string, which will play an important role later when the program is installed. The routine looks for this ID string within the handler for interrupt 09H. If it finds the string, the TSR program is already installed and can be reinstalled.

If the ID string isn't found, the TSR program hasn't been installed, or another TSR program redirected the interrupt 09H vector in the meantime. The TSR program can then be installed with the help of the installation routine. This routine must receive the ID string used to detect whether the program has already been installed, the address of the high level routine that will be called when the TSR program is activated, and the hotkey value. The hotkey value is the bit pattern, in the BIOS keyboard flag, that will activate the TSR program and can be defined within the high level language program with the help of predefined constants.

The initialization routine first saves the addresses of the interrupt handlers for interrupts 09H, 13H and 28H. Then the data for the context of the high level program are read and saved in variables within the code segment, so they are available for the interrupt handler and for activation of the TSR program. In the next step, the new interrupt handlers for interrupts 09H, 13H, and 28H are installed. Finally, the number of paragraphs after the end of the program, which are to remain resident, must be calculated. Here the C and Pascal modules differ from each other. Information about this calculation can be found in the individual descriptions of the modules.

The actual installation is now complete and the program is terminated as resident. Notice the installation routine doesn't return to the high level language program, so all initialization, such as memory allocation or variable initialization, must be performed before the call to this routine.

If the installation test function of the assembly language module determines the program is already installed, it can be reinstalled with the help of another function. This function is passed the address of a routine in the high level language program, which will perform a "cleanup" of the program. This process includes releasing allocated memory and other tasks. If no such routine is to be called, the assembly language routine must be passed the value -1. Since the "cleanup" function is in the TSR program, instead of in the program that is performing the reinstallation, a context switch is necessary. Unlike activation of the TSR program and the corresponding interruption of the foreground program, this is from the program that is performing the reinstallation to the already installed TSR program. The reinstallation returns the redirected interrupt handlers to their old routines and releases the memory allocated by the TSR program.

Besides these three functions, which are called from the high level language program, the assembler module contains some routines that may not be called by high level language programs. These include the interrupt handlers for interrupts 09H, 13H, and 28H as well as a routine that accomplishes the context switch to and from the TSR program.

Installation and setting the hotkey

The TSR program is usually installed after TsrIsInst has determined that a copy of the program hasn't been installed yet. The installation consists of two steps; the first step determines the program's hotkey using TsrSetHotkey. The program is then loaded into memory as a TSR program by the TsrInit function, and is, temporarily, terminated.

First, let's discuss TsrSetHotkey: The arguments used by this function are the two parameters that determine the hotkey. These are the bit mask for the modifier keys, and the scan code of the accompanying letter or number key. Both parameters can be constructed using constants presented at the beginning of the two programs in Pascal and C.

For the modifier keys, these parameters carry the names LSHIFT (left SHIFT key), RSHIFT (right SHIFT key), ALT, CTRL, etc. When the hotkey key combination must use more than one of these keys simultaneously, such as **Ctrl+Alt** with another key, they can be linked with a binary OR operator. This binary OR has the same effect for the user as the logical AND.

The constants for the keyboard scan codes all begin with the prefix SC_, which is then followed by the letter, number, or name of that key (for instance, SC_5, SC_X, or SC_SPACE). These constants can easily be found in both the Pascal and C program, since they are grouped into a large block. If your TSR program must be activated by a combination of only modifier keys, so no letter or number key must be pressed, you can use the SC_NOKEY constant.

Once the hotkey has been defined, TsrInit is called and transforms the program into a memory-resident TSR program. TsrInit also expects two arguments: the offset address of the actual TSR procedure and a value indicating the program's memory requirement. Since this parameter is handled differently in the Pascal and C assembly interfaces, we'll discuss it later. For now, simply remember the first parameter for TsrInit, which specifies the TSR procedure offset address, also determines the high level language procedure that is called when the TSR program is activated. This procedure embodies the actual purpose of the TSR program and is capable of utilizing all functions provided by the particular high level language that's used. It can even access files, read directories, and perform any other operation involving DOS functions. Once this procedure has been completed, the TSR program will again move to the background and clear the way for the previously interrupted program.

TsrInit's first task is to determine the addresses of interrupt handlers 08H, 09H, 13H, 28H, and 2Fh, and to store these addresses. Then the data required by the applicable high level language program are determined. These are also stored in variables within the code segment, so they'll be available to the interrupt handler as well as for activating the TSR program.

Next, the new interrupt handlers for interrupts 08H, 09H, 13H, 28H, and 2FH are installed. The function number that was specified at the previous `TsrIsInst` call is assigned as the function number for MUX interrupt 2FH.

Before the program can be installed in memory by the DOS function 31H, the amount of memory or the number of paragraphs the program will need to remain resident after execution must be calculated. The Pascal and C versions perform this task differently. Below, we'll explain how the two programs perform this task. This step completes the installation, and the program is terminated, but remains resident. Remember that once `TsrInit` has been called, the procedure doesn't return to the high level language program. Because of this, all tasks, such as memory allocation or the initialization of variables, must be completed before this procedure is called.

The high level language programs

The following programs in C and Pascal demonstrate the assembly language routines. First they determine whether the program is installed. On a new installation, a TSR routine is installed. You can activate the TSR by pressing both <Shift> keys. It stores the screen contents, then displays a message and asks the user to press a key. After this is done, the old screen contents are copied back and the execution of the interrupted program continues.

On a reinstallation, the assembly language reinstallation program calls a cleanup function in the TSR program. It prints the number of activations of the TSR program, which is set to zero when the TSR program is installed and incremented on each activation. This makes it clear the cleanup function is actually executed in the installed TSR program and not in the program that performs the reinstallation.

Removing TSR programs from memory

TSR programs are usually installed upon the first program call, and removed at the next call. So, if `TsrIsInst` determines, at the start of a program, that another copy is already resident, the resident copy must be removed.

For this, the `TsrCanUninst` function must be called. This function determines whether the program can be removed from memory because occasionally this won't be possible. This is the case when another TSR program has been installed since the original installation, because the second program also redirects the interrupt vectors of the timer, the keyboard, and other devices. This program, like other TSR programs, located the address of the first program's interrupt handler when it was first installed, and accesses these through its own handler. However, since the preceding handlers belong to the TSR program that must be uninstalled, these must also be removed. Since it isn't possible to inform the second program about removing these handlers, this action will inevitably result in a system crash.

Because of this, `TsrCanUninst` checks whether all redirected interrupts still point to the interrupt handler of the first copy of the TSR program, and responds with a corresponding TRUE or FALSE. Only when this function returns TRUE can the program call `TsrUninst` to remove the resident copy from memory.

For the uninstallation of a TSR program, the old interrupt handlers for interrupts 08H, 09H, 13H, 28H, and 2FH are restored. The memory occupied by the program is then released, so DOS can make it available to other programs. The program leaves no traces in memory.

Calling procedures in a resident TSR program

The possibility of using procedures within the resident copy of the TSR program must be used when another copy is reinstalled. This is because even the high level language portion of a TSR program must frequently use operating resources (memory, interrupt vectors, files) that must be returned to the operating system once the program is removed from memory.

Since the segment address of the resident copy can easily be determined through the MUX handler, and since the offset address of the corresponding procedure is the same as that of the program that was just executed, it's possible to construct a FAR pointer identifying the procedure of the resident program that will be executed. The only prerequisite for this operation is the routine to be called is of the type VAR, and that can also be arranged.

However, there's a problem with this procedure. By calling this procedure directly, the context isn't switched to the resident copy of the program. So, the data segment of the new running program, and its PSP and DTA, remain active. This means the

procedure being called couldn't access its variables stored in its data segment, because these belong to the copy of the program that was just called. Therefore, a mediator must be used when calling a procedure within the resident copy so the context can be switched to that copy of the TSR program. The same mediator would switch the context back to the program being executed once the procedure has been completed.

This type of mediating procedure would only require the offset address of the procedure within the resident program that must be called. Although this works, it doesn't provide a way to pass arguments to the resident procedure and return arguments from this procedure. To make the transmission of such arguments possible, another method of calling procedures within the resident program was selected. This method uses two procedures within the assembly interface: `TsrSetPtr` and `TsrCall`.

`TsrSetPtr` is called first. This procedure determines the address of the resident procedure that must be called and stores it in a variable of the assembly module. Then `TsrCall` is activated, which switches the context and uses the recorded address to call the desired resident procedure. Again, the parameters that must be passed to and from the resident routine present an obstacle, because `TsrCall` must be declared within the high level language module. After all, the number and types of parameters required depends on the particular procedure that is being called. However, as you'll learn in the description of the Pascal and C programs, this problem can also be solved.

The interrupt handler

The interrupt handlers of the assembly interface operate according to the principles illustrated above. The most prominent of these handlers is the keyboard interrupt handler, which serves interrupt 09H. The coordination between this particular handler and the interrupt handlers for interrupts 08h (timer), 13h (BIOS disk), and 28h (DOS idle) is managed by three flags in the code segment of the assembly interface: `in_bios`, `tsraktiv`, and `tsrn timer`.

Flags

The `tsraktiv` flag indicates whether the TSR program is currently active. It carries either the value 0 or 1. The same applies to `inbios`, which is incremented upon accessing handler INT 13H and is decreased upon leaving the handler. Since `inbios`, like all other flags, is initialized at 0, it carries the value 1 during an INT 13H call and then returns to 0. However, when another function of interrupt 13H calls the interrupt recursively, this flag is further incremented, increasing its value. However, it's important the flag carry the value 0 when no BIOS disk function is currently being executed. This indicates that at least this path is clear for the TSR program start.

The third flag, `tsrn timer`, is set within the keyboard interrupt handler when the hotkey code has been detected and the TSR program currently cannot be activated, because of the reasons explained above. This flag is then checked by the timer and DOS idle interrupt handlers, to determine whether the TSR program is waiting to be executed.

The timer interrupt handler decreases the value of `tsrn timer` with each call, so it will eventually reach 0 and the attempt to activate the program will be discontinued (since the user believes the program will never start). The code for the different interrupt handlers shows several interesting details of interrupt handler programming. We're particularly interested in the two handlers for timer interrupt 08H and keyboard interrupt 09H. Since the interrupt handlers of the assembly interfaces in both Pascal and C are identical, the discussion applies to both of these modules.

The timer interrupt handler

The following excerpt from the code of the assembly module contains the new interrupt handler for timer interrupt 08H. The first command already checks the `tsrn timer` flag. Interestingly, the command doesn't include a segment override (i.e., `cmp cs:tsrn timer,0`), because this variable is found within the code segment. However, this override doesn't have to be included in the source code, because the assembler automatically includes it when the code is assembled into machine language. Also, an `ASSUME` command was used to indicate that only segment register CS is pointing to the code segment, and the contents of the DS register, as well as all other segment registers, are unknown.

```
assume cs:code, ds:nothing, es:nothing, ss:nothing
```

So, all interrupt handlers can access the different variables and flags of the assembly interface without having to explicitly specify a segment override. All of these have been stored in the code segment, so the data segment of the corresponding high level language program doesn't have to be continually loaded.

Now let's return to the timer interrupt. If the handler discovers, from the comparison of the `tsrnow` flag with 0, the TSR program isn't waiting to be activated (`tsrnow = 0`), it immediately jumps to label `i8_end`. There, the previous interrupt handler of the timer interrupt is called by the command:

```
jmp [int8_ptr]
```

`Int8_ptr` isn't a label, but rather a variable, in which the address of the previous interrupt handler was recorded at the installation of the TSR program using `TsrInit`. Since `int8_ptr` is a `DWORD` variable, the assembler knows that a `FAR-JMP`, instead of a `NEAR-JMP`, must be executed. After all, the previous handler is located in a different code segment than the currently active interrupt handler (belonging to the TSR program) and can be reached only in this way.

The `IRET` command found at the end of the old interrupt handler returns the execution to the program that was suspended by the call the `TIMER` interrupt handler. So, the jump to the old handler ends the execution of the new handler from within the assembly interface.

```

;-- New interrupt 08h handler (timer) -----

int08      proc var

            cmp     tsrnow,0           ;is TSR to be activated?
            je      i8_end             ;no, continue to new handler

            dec     tsrnow             ;yes, decrease incrementation flag

            ;-- TSR is to be activate, but is this possible? -----

            cmp     in_bios, 0         ;is BIOS disk interrupt currently active?
            jne     i8_end             ;YES--> cannot activate

            call    dosaktiv           ;may DOS be interrupted?
            je      i8_tsr             ;yes, call TSR

i8_end:     jmp     [int8_ptr]         ;jump to old handler

            ;-- activate TSR -----

i8_tsr:     mov     tsrnow,0           ;TSR is no longer waiting to activate
            mov     tsraktiv,1         ;TSR will activate shortly
            pushf                     ;simulate call of old handler using
            call    [int8_ptr]         ;INT 8h command
            call    start_tsr          ;start TSR program
            iret                     ;return to interrupted program

int08      endp

```

However, what happens when `tsrnow` isn't equal to 0, which indicates the TSR program is waiting to be activated? The value of this flag will then be decreased, so possibly even at the next call the `TIMER` interrupt handle the flag's value will have reached 0. At that point, the attempts to start the TSR program are discontinued so the program isn't activated after a several second delay.

First, however, the TSR program must be activated so the next step consists of testing whether this is possible. For this, the `in_bios` flag is first read. If this flag contains a value other than 0, then the timer interrupt has just interrupted a ROM BIOS disk operation in progress. If so, the TSR program isn't activated and the execution jumps back to the old timer interrupt. If the flag's value is indeed 0, the `INDOS` flag is checked to test whether a DOS function has been interrupted during its execution.

For this check, the assembly procedure is called. This procedure simply reads the address of `INDOS` from a variable within the assembly module and then compares the value at this address (the `INDOS` flag) with 0. If this test indicates that DOS cannot be interrupted at this time (when `INDOS > 1`), execution returns to the old interrupt handler and the TSR program isn't activated. If this test is passed as well, nothing can stop the TSR program from being activated. Before the actual start, however, the old interrupt handler of the timer interrupt must be executed. This handler plays an important role in time keeping and disk access. This handler also contains a machine language command that is very important to all interrupt handlers that are linked with hardware interrupts, specifically the following command:

```
out 20h,20h
```

This command informs the interrupt controller the interrupt's execution has been completed. The interrupt controller won't trigger hardware interrupts until this command is executed. This would not only mean that all further timer calls would drop out, so the PC's internal clock would freeze, but also the PC could receive no further keyboard input, since the keyboard interrupt 09H would be blocked.

The address stored in the variable `int8_ptr` is used to call the old interrupt handler. However, here a `CALL` command is used, so the CPU returns to the TSR program's new handler upon encountering the `IRET` command. For this to occur, the contents of the flag register must be pushed onto the stack, using `PUSHF`, before `CALL` is executed. Usually `CALL` places only the return address on the stack, instead of the flag register, as is the case with an interrupt call. As a result of this, `IRET` would read part of its supposed return address from the flag register, and obtain the return address from the stack, which contains completely different information than was stored there earlier. This mixup results in a system crash.

Although the TSR program is started only after the old interrupt handler has been called, the `tsrnw` and `tsractive` flags are set to 0 and 1, respectively, before this point. Basically this indicates the TSR program is already active, and is therefore no longer waiting to be started. This is justified because the old handler informs the interrupt controller the execution of the interrupt has been completed (see above). This enables further interrupts to be processed while the timer interrupt handler is being executed. For example, a keyboard interrupt should be executed if the user pressed another key in the meantime.

If this keyboard input happened to be the program's hotkey, the TSR program would be activated again by the keyboard interrupt handler if `tsractive` wasn't already set to 1. This results in another unacceptable situation: the execution of the new timer interrupt handler could begin only after the TSR program has been terminated again, which means the program would start again immediately afterwards. However, since the proper flags have been set before the old handler is called, the other interrupt handlers are unable to call the TSR program at that time. This ensures the problem we explained above doesn't occur.

Finally, `start_tsr` is called to activate the TSR program. This is an assembly procedure within the assembly module, which first secures the context of the current program that is being interrupted, sets the context of the TSR program, and then calls the TSR procedure that was specified in combination with `TsrInit`. After the TSR program has been completed, the context of the interrupted program is restored, and the procedure is ended.

Only then does the program execution return to the new timer interrupt handler, which finally returns to the interrupted program using `IRET`. Interestingly enough, the new timer handler is called repeatedly during execution of the TSR program, although its own execution basically hasn't been completed yet. However, this remains without consequence, providing the address of the original, interrupted program is still located on the stack upon the return from `start_tsr`. Also, to the system, the execution of the new timer interrupt was completed the moment the "out 20h,20h" was executed.

The keyboard interrupt handler

Much of what has been said about the timer interrupt handler also applies to the keyboard interrupt handler, serving interrupt 09H. Here, the `AX` register contents are first saved to the stack, since this register will be changed within the handler. Remember that once it's completed its execution, an interrupt handler cannot leave any of the processor registers changed.

This didn't cause problems for the timer interrupt handler because it didn't change any registers. However, this must be considered for the keyboard interrupt handler.

Contents of port 60H are read in the next step. The keyboard controller will store the scan code of the key, which the user pressed, in this port. The new keyboard handler must examine this scan code to identify the hotkey. First, however, this new keyboard handler checks whether the TSR program is already active. If so, all further tests are skipped. The contents of the AX register are then pulled off the stack. The execution jumps directly from label i9_end to the old keyboard interrupt handler.

The same steps are taken when tsractive is 0 and tsrnow carries a value unequal to 0. This indicates the hotkey has already been detected and the TSR program is waiting for an opportunity to activate. So why detect the hotkey a second time? This happens when the TSR program is neither active, nor waiting to be activated. Therefore, the hotkey scan code is first read from the variable sc_code. If the value contained in this variable is 128, a real hotkey hasn't been defined, and only the status of the modifier keys must be checked. In this case, the execution jumps to label i9_ks.

```

;-- New interrupt 09h handler (keyboard) -----

int09      proc var

            push ax
            in  al,60h                ;read keyboard port

            cmp  tsraktiv,0           ;is the TSR program already active?
            jne  i9_end               ;YES: call old handler, then return

            cmp  tsrnow,0             ;is the TSR waiting for activation?
            jne  i9_end               ;YES: call old handler, then return

            ;-- test for hotkey -----

            cmp  sc_code,128          ;is a scan code defined?
            je   i9_ks                ;No, check modifier {{?}} keys

            cmp  al,128               ;if yes, is it the release code?
            jae  i9_end               ;yes, but it is not the hotkey

            cmp  sc_code,al           ;make code, compare with key
            jne  i9_end               ;not correct, do not activate

i9_ks:      ;-- check modifier key status -----

            push ds
            mov  ax,040h              ;pull DS to the variable segment
            mov  ds,ax                ;of ROM-BIOS
            mov  ax,word ptr ds:[17h] ;get BIOS keyboard flag
            and  ax,key_mask           ;screen out non-hotkey bits
            cmp  ax,key_mask          ;are only hotkey bits remaining?
            pop  ds
            jne  i9_end               ;hotkey detected? NO --> return

            cmp  in_bios, 0           ;BIOS disk interrupt currently active?
            jne  i9_e1                ;YES --> cannot activate

            call dosaktiv              ;may DOS be interrupted?
            je   i9_tsr                ;yes, start TSR

```

```

i9_e1:      mov    tsrnow,TIME_OUT    ;TSR waiting to be activated

i9_end:     pop     ax                ;get AX back
            jmp     [int9_ptr]        ;jump to old handler

i9_tsr:     mov     tsraktiv,1        ;TSR active (in a second)
            mov     tsrnow,0          ;no delayed start wanted
            pushf
            call    [int9_ptr]        ;call old handler
            pop     ax                ;get AX back
            call    start_tsr         ;start TSR programs
            iret                     ;return to interrupted program

int09      endp

```

If a hotkey has been defined (sc_code not equal to 128), the scan code of the key that's been pressed is read from port 60H. If this code is larger than 128, it's identified as a release code, which indicates that a key has been released, instead of pressed. Since the pressing, instead of the releasing, of a key must start the TSR program, the key code isn't analyzed further. The execution then immediately jumps to the old handler.

Assuming that a key has been pressed, it's code is then compared with the value stored in sc_code. If these values don't match, the key is identified as not being the hotkey, and the execution jumps to the old keyboard interrupt handler. However, if the hotkey code matches the specified value, the status of the modifier keys must be checked. With this, we've returned to label i9_ks, which was mentioned above. Here, the current modifier key status is read from the BIOS variable at address 0040h:0017h and compared with the value stored in key_mask. If the two values don't match, execution returns to the old keyboard interrupt handler, which then takes over the task of processing the keyboard input.

If this comparison determines the specified keys have been pressed, the TSR program must be activated. Before this can occur, you must check whether a BIOS disk interrupt or DOS API function is currently being executed. If any such function is currently active, the program isn't activated and execution returns to the old keyboard handler. However, before this jump, the tsrnow flag is set to the value specified by the TIME_OUT constant. This delay value gives the TSR program a chance to be activated during one of the next timer interrupt calls.

In the code listing provided, TIME_OUT is assigned the value 9, which corresponds to half of a second, since the timer interrupt is called 18.2 times each second. However, you may replace this value with one of your own values if you want to increase or decrease the maximum delay time in which the TSR program can be started after pressing the hotkey.

If DOS or BIOS functions aren't preventing the activation of the TSR program, it's necessary, as was the case with the timer handler, to call the old keyboard interrupt handler. Here again, the tsractive and tsrnow flags are initialized with appropriate values so no other interrupt handler will be able to activate the TSR program during this time. This then occurs automatically within the keyboard interrupt handler once the old handler is called.

The Pascal and C programs

Now we'll discuss the high level language programs, TSRP.PAS and TSRC.C, which demonstrate the use and function of the two assembly modules. Their structure is almost identical; they have several differences, which we'll discuss below. First, let's concentrate on the common characteristics of these two programs. Upon program start, the ParamGetHotKey function evaluates any parameters entered on the command line. It accepts all parameters preceded by the prefix "/" as hotkeys. This prefix may be followed either by the name of a modifier key (LSHIFT, RSHIFT, ALT, CTRL, etc.) or the scan code number of a particular key. This number must be entered as a decimal value.

To specify the left **Shift** and **Spacebar** as the hotkey key combination, the following two parameters must be entered (the order in which these parameters are entered isn't important):

```
/tlshift /t57
```

If **Alt** must be included in this hotkey, the following parameters would be required:

```
/tlshift /talt /t57
```

ParamGetHotkey then stores the specified modifier key status and scan code in the two variables KeyMask and ScCode, which are passed to the function for this purpose.

If an error is discovered when the command line entry is evaluated, the program execution is stopped and a corresponding error message is displayed. If the parameters were entered correctly, the TsrIsInst function then checks whether the program has already been installed. Here, the function number through which the program's MUX handler can later be reached is also determined. Although the constant I2F_CODE specifies the function C4H for this, you can select another function. You must do this if you develop more than one TSR program using these assembly interfaces; otherwise these TSR programs would use the same MUX function and therefore conflict with one another. However, you must be careful when selecting other MUX function numbers since some of these may coincide with the function numbers of existing programs. However, don't use values smaller than C0H. See Chapter 28 for more information about the DOS multiplexer.

If TsrIsInst indicates the program hasn't been installed yet, the values of KeyMask and ScCode are checked to determine whether /t parameters were entered from the command line. If these parameters weren't entered, the variables will contain default values. In this case, the TsrSetHotkey assembly procedure will define **Alt**+**Q** as the hotkey key combination. If /t parameters have been entered, then TsrSetHotkey sets this specified hotkey combination. Now the only remaining procedure to be executed is TsrInit, which transforms the program into a TSR program. The high level language procedure TSR is thereby specified as the TSR procedure. We'll discuss this in more detail later.

If TsrIsInst indicates the program has already been installed, then the following action depends on the original user input. If a hotkey was specified at the program call, TsrSetHotkey will install the specified key combination as the updated hotkey in the resident program, then the program is ended without performing a new installation. However, if a hotkey wasn't specified, TsrCanUninst is first used to check whether the resident copy may be deactivated. If it can, TsrUninst is called to remove the resident copy from memory.

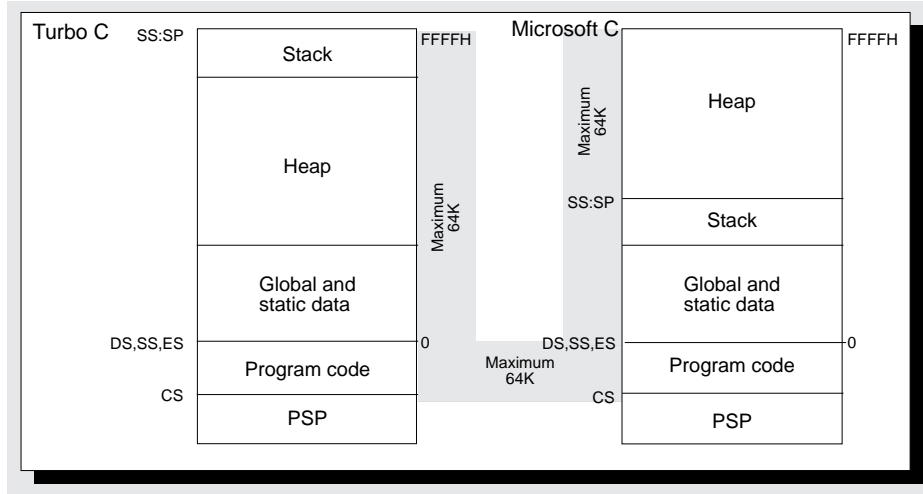
Before the TSR program is uninstalled, however, a high level language routine named endFCT (in C) or EndTPrc (in Pascal) is called. This routine frees the program's internal resources and displays a message indicating the number of times the TSR program was activated. This number is defined by the global variable ATimes, which is incremented each time the TSR program is activated. This is done in the program's TSR procedure named Tsr. Before the TSR program is removed, then, the keyboard buffer is emptied to remove the hotkey, the screen of the previously interrupted program is saved, and a message asking the user to press any key is displayed. As soon as the user has done this, the screen of the interrupted program is restored, and the TSR program procedure ends. Then the execution of the previous program resumes.

The C implementation

Since TSR programs should use as little memory as possible, the assembly language interface was developed to be linked with the smallest C memory model (the small model). In both Microsoft and Turbo C compilers, the program code and data are placed in two separate segments, each of which cannot be larger than 64K. The data includes global and static data as well as the stack and the heap. As the next illustration shows, Turbo C and Microsoft C use different memory organization, despite their similarities. In Turbo C the stack is placed behind the heap and moves from the end of the data segment to the end of the heap and in Microsoft C the stack is between the global data and the heap.

If this organization doesn't affect the assembly language interface, we could allocate the entire 64K of the data segment resident in memory in addition to the program code. However, since this would waste a lot of memory and TSR programs should use as little memory as possible, only the part of the data segment that is actually needed should be marked as resident by the assembly language module.

Structure of a small model program (Turbo C and Microsoft C)



The size of this memory area depends on the size of the data (or objects) that will be allocated on the heap by the functions `calloc()` and `malloc()`. You must guess this size and pass it to the initialization routine so the end of the required memory in the data segment can be calculated. This mechanism allows you to use the heap functions normally within the TSR program. Unfortunately, this applies only to the Turbo C compiler. Microsoft C uses an allocation algorithm that assumes that all the memory to the end of the data segment is available. So, allocating heap storage should be avoided within a TSR program compiled with Microsoft C. You should allocate the buffers and variables required when the TSR program is initialized or place the required objects in global variables. The example C program allocates the two buffers it needs in the `main()` function and then places the addresses of the buffers in global variables.

There is something else you should be aware of when using Turbo C. Since the stack grows from the end of the 64K data segment to the heap, it finds itself outside the program when parts of the data segment are released again, and in an area of memory that DOS may give to other programs. To avoid problems with this, the assembly language module places the stack immediately after the heap, giving it 512 bytes of space. This should be sufficient for most applications, but may lead to problems if you use large objects (such as arrays) as local variables or pass them to other functions using the stack. In this case, you should enlarge the stack by setting the constant `TC_STACK` in the assembly language module to a larger value.

Because of this differentiated use of the stack, `TsrInit` must be informed whether the program has been developed using a Microsoft or a Borland compiler. However, since this is managed from within the C program with constants that are defined through conditional preprocessor instructions, you don't need to worry about this operation.

Such functions are also used by `GetHeapEnd`, which is called from the assembly module by `TsrInit`. It supplies the assembly module with a FAR pointer to the end of the occupied heap. With Borland compilers, this information can be gathered through the library function `SBRK`, which is also the case in MSC up to Version 6.0 and in QuickC up to Version 2.5. However, in current versions, this function is no longer supported. So, the library function `_heapwalk` must be used to search the stack for the last occupied heap block.

The way in which the C version of the program calls functions within the resident copy of the program is also interesting. As we mentioned in the description of the assembly module, the assembler procedure `TsrSetPtr` must first be called. This procedure receives the address of the procedure that must be executed and stores it. The stored address is later accessed by `TsrCall` when this function is called. In the C version of the assembly module, `TsrSetPtr` then directly returns a pointer to `TsrCall`. The advantage of this is the desired function can be called directly using the result of `TsrSetPtr`. However, this requires a CAST operation to secure the compiler's cooperation. Let's take a closer look at this operation.

At the start of the C module, two types of function pointers, `OAFP` and `SHKFP`, are defined. Of these two, `OAFP` points to all function types that don't require arguments, and therefore also don't return function results. `SHKFP`, however, has been tailored specifically to meet then requirements of the `TsrSetHotkey` assembler procedure.


```
typedef void (*OAFP)(void);
typedef void (*SHKFP)( WORD Keymask, BYTE ScCode );
```

The following expression is used to call TsrSetHotkey using TsrSetPtr:

```
(*(SHKFP) TsrSetPtr(TsrSetHotkey))( Keymask, ScCode );
```

This expression first calls TsrSetPtr, in which the address of TsrSetHotkey is passed, as one of its arguments, in the form of a FAR pointer. Actually, this operation requires only a NEAR pointer, since the segment address of the resident copy must later be used as the segment address anyway. However, the various functions that are called in this way must be FAR so they can be called out of other code segments. To avoid the compiler error message following the otherwise impending transformation of a FAR pointer to a NEAR pointer, TsrSetPtr receives the FAR address of the desired function and processes only its offset address.

TsrSetPtr then returns the address of TsrCall as a NEAR pointer. However, in casting the pointer, the expression from above determines that this pointer is of type SHKFP. This is the only technique where the desired parameters can be entered so the compiler will copy them to the stack without any interjections. TsrCall is finally called by referencing the pointer cast in this way. Therefore, when TsrCall is executed, it finds the arguments, which are actually intended for the function that will be called, located on the stack. These arguments must be copied to the stack of the resident copy of the TSR program, since we must switch to this stack before this function can be called. This is because most C compilers generate code under the assumption the segment register DS point to the same memory segment as register SS. If the stack wasn't switched, this assumption wouldn't be true, and the function couldn't be executed properly.

Therefore, if you want to use the same technique for calling functions in a resident copy of your TSR program, you'll need to follow these two steps:

1. Define a function pointer type that emulates a function with arguments that are also required by the actual function you want to call.
2. Pass the address of this function to TsrSetPtrs, cast the function result in a previously defined pointer, and call the function using the required arguments.

You'll find the following program(s) on the companion CD-ROM

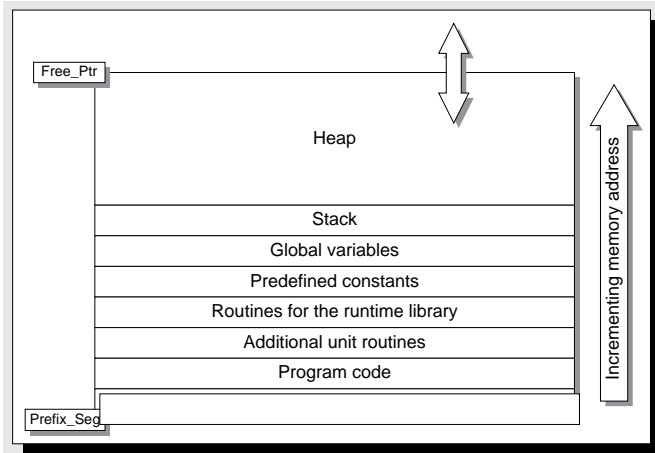


TSRC.C (C listing)
TSRCA.ASM (Assembler listing)

The Pascal implementation

Turbo Pascal offers only one memory model, unlike the various C compilers. The organization of this model is well suited to TSR programs. The illustration below shows the program code and the required routines from the various units and the

Memory layout of a Pascal program under Turbo Pascal 4.0



runtime library follow the PSP. After these are the predefined constants, the global data, and the stack segment. While the size of these program components are set at compilation and cannot be changed after the program is loaded into memory, this doesn't apply to the size of the heap, which follows the stack segment. When new objects are created with the NEW command, the heap grows toward the end of memory.

Unlike C compilers, Turbo Pascal allow you to set the maximum size of the heap, as well as the stack size, with a compiler directive inside the source code. This is the \$M directive, which must be passed the following parameters:

```
{ $M stack size, minimum heap size, maximum heap size }
```

All specifications are in bytes, so the directive:

```
{ $M 2048, 0, 5000 }
```

results in a 2K stack and a maximum 5000 byte heap. If no such directive is found in a program, the heap isn't limited and it can grow to the end of main memory. This would have devastating results for a TSR program, because the entire memory must be reserved for the TSR program and there would be no memory left for additional programs. But with the \$M directive placed at the beginning of the program, we can set the maximum size of the program in memory and the number of paragraphs that must remain resident after the program is terminated.

Turbo Pascal also allows the number of paragraphs to be reserved to be calculated from the Pascal program, which eliminates the complicated calculation in the assembly language interface. In a C program, important data needed for this calculation (segment addresses of the PSP and data segment, and size of the heap) are available only at the assembly language level, but Turbo Pascal places this information in normal variables, which are available to a Pascal program in the form of pointers. For our purposes, we need the starting address of the PSP and the end of the heap, since they mark the start and end of the TSR program in memory.

The figure shows the segment address of the PSP is found in the variable PrefixSeg, while the end of the heap (up to Version 6.0) is determined with the help of the pointer variable FreePtr. This variable doesn't point directly to the end of the heap, but the segment portion of this pointer contains the end address of the heap minus \$1000. This information is used within the TSR program in the ResPara procedure, which calculates the number of paragraphs to remain resident after the installation of the TSR. Version 6.0 changes this; the HeapEnd pointer directly indicates the end of the heap.

The procedure ResPara within the TSR program uses this information by utilizing these variables to calculate the number of paragraphs the TSR program will occupy while remaining resident in memory. Through the help of conditional compiling, either the HeapPtr or the HeapEnd pointer can then be accessed for this purpose, depending on the Turbo Pascal version used.

However, the procedures, within the resident program, that will be executed must be of the type FAR. Unfortunately, calling these functions isn't as simple here as in the C version, since Turbo Pascal doesn't permit the casting of function pointers. Because of this, TsrSetPtr doesn't return a result upon its call, and the calls of TsrSetPtr and TsrCall cannot be combined.

However, in the Pascal version, it's also necessary to declare code pointers that depict the procedures and functions that will be called, and particularly their arguments. As you can see in the following listing of TSRP.PAS, these pointers are called WoAPrcK and SHKPrck. WoAPrcK is a pointer to a procedure that doesn't require any arguments, while SHKPrck is specifically designed to serve the TsrSetHotkey procedure.

```
type WoAPrcK = procedure;           { Procedure without arguments }
     SHKPrck = procedure( KeyMask : word;           { TsrSetHotkey }
                          ScCode  : byte );
     PPtrT = record                { Union for creating the procedure pointer }
       case integer of
         1 : ( WoAPrc : WoAPrcK );
         2 : ( SHKPrck : SHKPrck );
       end;

const Call : PPtrT = ( WoAPrc : TsrCall );
```

These types are collected in a variant record that contains an entry for each of these types. Here these entries have been named WoAPrc for WoAPrcK and SHKPrck for SHKPrck. A global variable named Call has been declared for calling the functions that are associated with these types. The WoAPrc component of this variable is initialized as a pointer to the TsrCall procedure.

This variable can then be used to call the desired procedure or function, providing its offset address is first passed to TsrSetPtr. This is illustrated by the following two program lines. At the TsrSetPtr function call, TsrSetHotkey is specified as the procedure that will be executed. TsrCall is then called together with the arguments for TsrSetHotkey.

```
TsrSetPtr(ofs(TsrSetHotKey));  
Call.SHKPrC( Keymask, ScCode );
```

This approach works because, by employing the SHKPrC component of Call, the compiler assumes that such a procedure is actually being called. Actually, however, TsrCall is being executed. In respect to the stack, this function has it easier than its C counterpart.

The reason for this is that Turbo Pascal programs don't expect the data segment to correspond to the stack segment. So, the stack of the currently running program can remain active, even when a procedure or function within the resident copy of the program is being called.

Tips on using TSR programs

In this chapter we've demonstrated how easily TSR programs can be developed, even in high level languages, by simply placing the core procedures in assembler modules and linking these with high level language modules. However, TSR programming is still a complicated task.

So, we recommend using several techniques that are suitable for TSR programs. First develop the program as a normal program that can be compiled from DOS, or an integrated development environment, and executed.

You should also try to reduce the number of changes needed to convert the program to a TSR program. To do this, you should also develop an initialization procedure, as well as the actual TSR procedure, which will be called when the hotkey is activated. However, unlike the final TSR program, these procedures should still be called from within the main program procedure (or function) so their execution doesn't depend on the hotkey yet.

In this way, you can completely develop and test the program. Once you're satisfied with the results and have found all its bugs, you can convert it to a TSR program. Correcting all errors prior to this is particularly important, since it's almost impossible to detect errors once the conversion has occurred, even through the use of a debugger.

The actual conversion into a TSR program is actually rather simple. You only need to integrate the assembly module with the program, and to call the appropriate functions from the assembler module. The two sample programs illustrate the logistics of this operation in detail, and all necessary function calls can be found within the main program.

You'll find the following program(s) on the companion CD-ROM



TSRP.PAS (Pascal listing)
TSRPA.ASM (Assembler listing)



Protected Mode, DOS Extensions, DPMI/VCPI

Recently, 80286 Protected mode has been making headway into the world of DOS. However, DOS is a Real mode operating system. So, it cannot use Protected mode. This is also true for the ROM-BIOS of a PC, which is also intended for Real mode operation. ROM-BIOS crashes when the first BIOS call is made after switching to Protected mode.

The 80386 and 80486 demonstrate their true power in Protected mode. Many software developers have been searching for a back door to make Protected mode usable with DOS. This has resulted in the creation of EMS emulators, memory management programs, DOS extensions, and multitaskers. These utilities run DOS as a Protected mode operating system. In this chapter we'll explain how Protected mode works and how it can be distinguished from Real mode.

Protected Mode

Protected mode was introduced with the 80286 microprocessor in 1982. It was originally developed for multitasking operating systems. However, since DOS is based on the older Real mode and would have to be completely rewritten to use Protected mode on Intel processors, it cannot use this mode. Almost ten years passed before Protected mode could actually be used. This occurred with the introduction of the 32-bit version of Windows and OS/2 Version 2.0. Previously, temporary solutions, such as the standard and enhanced modes used with Windows 3, were used.

Characteristics of multitasking operating systems

To understand the various characteristics of Protected mode, we must study how these characteristics are applied. This involves the structure and requirements of multitasking operating systems and how they are related to the capabilities of the underlying processors. Protected mode was developed to interact with this type of operating system.

The most obvious characteristic of multitasking operating systems is their ability to run several programs or tasks simultaneously. Not only do different programs run simultaneously, but often separate executable files within these programs also run. One example of this is a word processing program that formats a file and sends it to the printer while you continue entering text. These two tasks run concurrently within a single program. When executing programs concurrently in a multitasking environment, both tasks must be able to coexist in memory. Each task assumes that it's the only one controlling the computer. The tasks must operate in unison, while the operating system itself must be protected from the programs and their various tasks. This is why the name "Protected mode" was chosen.

Processor requirements for a multitasking environment

A multitasking environment must meet the following criteria:

1. Mutual protection of tasks and the operating system from overwriting areas of memory.
2. Support during task switching, particularly when restoring the executable state of a task.
3. Privileged status for the operating system when executing specific assembly language instructions and operations.
4. Support during the setup of virtual memory management.

We'll discuss how tasks and the memory areas are protected from each other later. However, remember that this is the first requirement imposed on an operating system.

Task switching support

The simultaneous execution of multiple programs in a multitasking environment is usually an illusion. Theoretically, true multitasking exists only when the hardware contains several processors, performing several tasks in parallel. Actually, usually only one processor is available at any given time. So, multitasking normally means that several tasks are sharing processor time, with each task using the processor for only a fraction of a second.

For example, suppose that a very slow system is running three tasks concurrently. The first task executes for a third of a second, then the second task executes for a third of a second, then the third task executes for a third of a second. Then the cycle repeats. This is called *preemptive multitasking* and follows the principle of *time slicing*. The time slicing process is based on an abrupt intrusion into task execution. After a specific period of time, the operating system simply stops executing the current task and continues with another task. This procedure must be transparent to the interrupted task. The task being interrupted cannot prepare for this interruption, because it can occur at any moment.

The operating system is responsible for ensuring that program execution continues without interruption between tasks. System resources (i.e., memory, files being processed, and processor registers), must remain unchanged by the task switch. Maintaining system integrity following an interruption can be managed through software, although task switching speeds up if the processor manages this assignment.

Finally, the illusion of parallel execution occurs only when switching between tasks is repeated many times per second. However, this switching wastes a lot of processor time and doesn't help task execution. So, support for task switching is the second requirement imposed on a processor with a multitasking capability.

Operating system privileges

The third requirement is the operating system must have certain privileges; priority must be given to certain assembly language instructions, while task execution remains on hold. These instructions include those involving task switching and those that influence the processor's operating mode. Suppose that a task simply switched the processor from Protected mode to Real mode. This probably wouldn't result in a system crash, just as overwriting some areas of memory wouldn't cause a crash. However, data and other programs could be corrupted. The operating system is responsible for correcting these errors or, if a correction isn't possible, ending the offending program and removing it from memory. Other programs should remain undisturbed.

Virtual memory

The processor should help the operating system manage virtual memory. Memory requirements increase with the number and complexity of the programs being executed simultaneously. Often the memory requirements exceed the amount of physical memory that's available. The operating system should allow programs access to more memory than actually exists, by using virtual memory. The memory areas that aren't needed by the current task can be moved to the hard drive until they are needed again. The processor supports the capacity to determine which memory is needed and which isn't.

Now let's see how these requirements are actually used. We'll look at Protected modes on the 80286, 80386, and 80486. The latter two Protected modes are downwardly compatible with the 80286, but include significant improvements. You'll see how efficient and how complicated programming in this mode can be. Unfortunately, all Intel processors work in Protected mode slower than in Real mode. In the following sections we'll explain why this occurs.

80286 Protected mode

Anyone familiar with assembler programming in Real mode under DOS, but who hasn't worked in Protected mode, often asks how the differences between the two modes affect the processor. If the processor suddenly no longer recognizes such familiar instructions as JMP, PUSH, or MOV in Protected mode, you may be wondering whether the processor then recognizes new instructions. Some programmers also fear that since they are confronted with a different processor, programming concepts used in Real mode no longer apply.

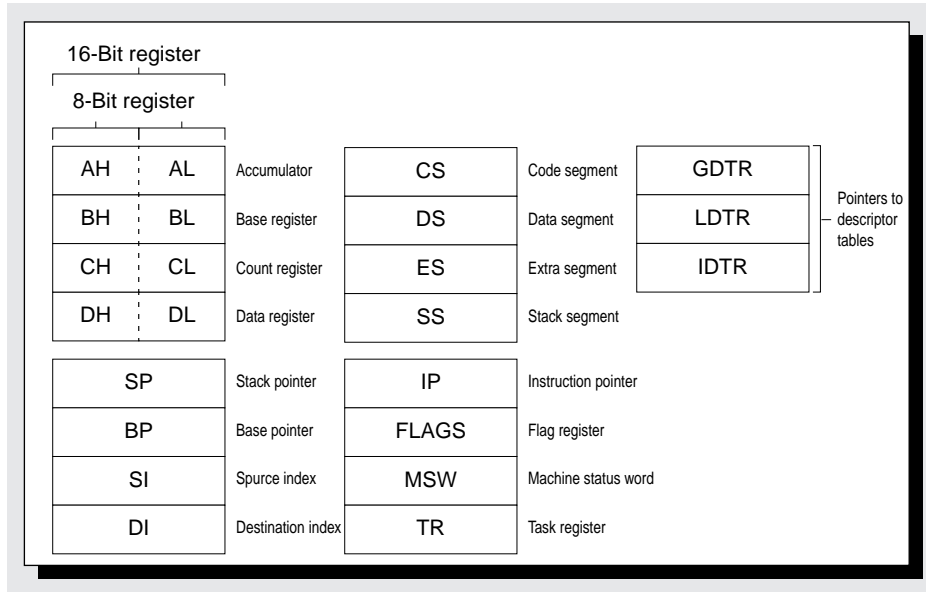
However, this isn't true. Instructions used in Real mode are also used in Protected mode; only a few new instructions have been added. However, the assembly language programmer barely notices many of the instructions. These include memory

addressing, such as loading of segment addresses in the segment register, and the creation and coding of FAR pointers within JMP instructions and subroutine calls.

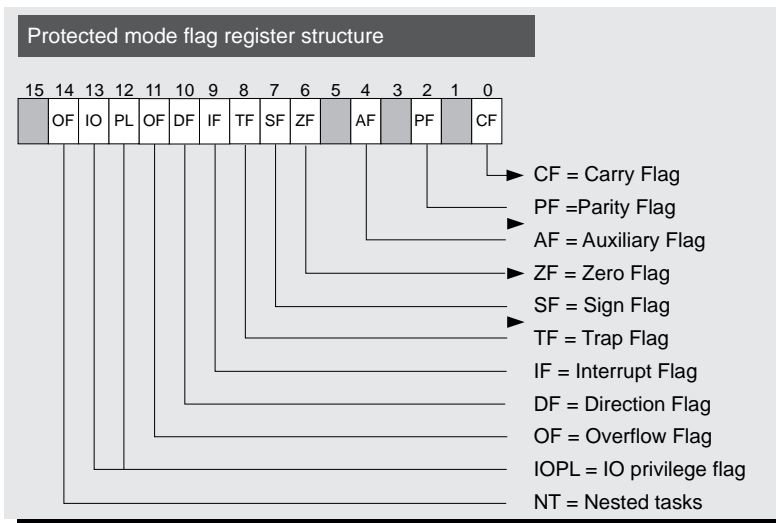
80286 registers

The register complement of the 80286 was expanded from the 8086 instruction set, as shown in the following diagram. Although these new registers are also available to the programmer in Real mode, they have no meaning. Some of the registers, such as GDTR, LDTR, and IDTR, must be loaded before switching to Protected mode because the system relies on these registers. We'll discuss the meaning of these registers below.

The 80286 registers in Protected mode

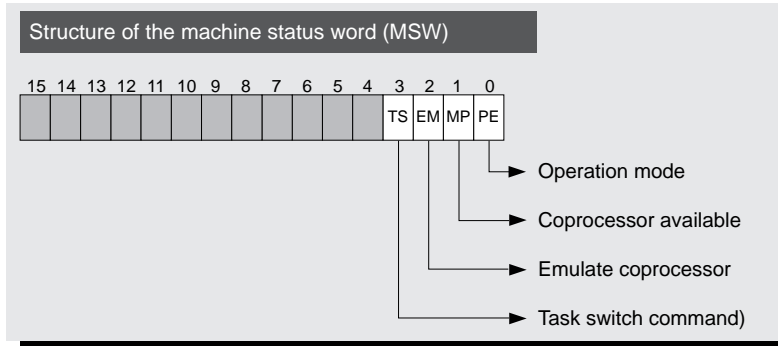


The Protected mode uses the flag register in almost the same way as Real mode. Existing flag positions remain unchanged, but Protected mode adds two new flags (IOPL and NT). Both flags are discussed in detail later in this section.



Switching to Protected mode

The machine status word (MSW) is a type of flag register because the individual bits describe the processor status. The 80286 processor uses only the lower four bits of the 16 available bits. Bits 1, 2, and 3 (MP, EM, and TS) support a math coprocessor and aren't important in the context of this chapter. However, bit 0 (PE) is the key to the Protected mode. When the processor is initialized, this bit is set to 0, which places the processor in Real mode. If a program loads the value 1 into this bit, the processor returns to Protected mode.



Unfortunately, resetting this bit doesn't automatically return the 80286 to Real mode. Eventually DOS programs, such as DOS extensions or EMS simulators, always reach a point where they must switch back to Real mode, when enables the user to continue working under DOS. Complicated procedures are needed to do this because the 80286 doesn't have an instruction for resetting the PE bit in the machine status word. Usually the ROM-BIOS must be initialized, which clears all RAM. Various tricks must be used to prevent this from happening.

It's unknown why the 80286 isn't able to restore Real mode. However, switching the processor back to Real mode from Protected mode is possible. Unfortunately this process is time-consuming and requires some elaborate precautions. The task register (TR) manages and switches between the various tasks (more on this later).

Memory management in Protected mode

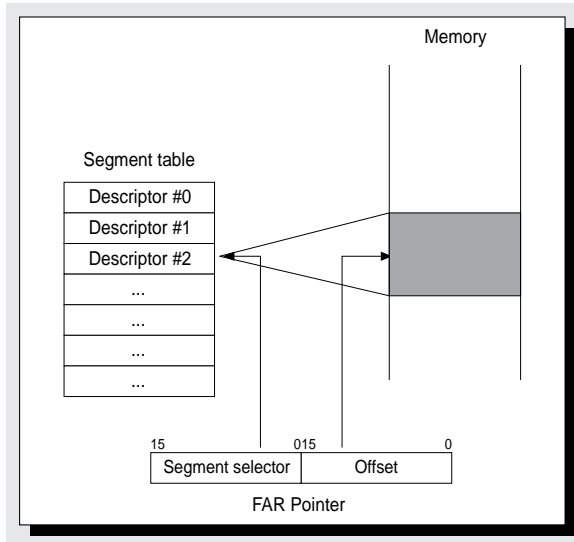
Real mode offers one megabyte of address space. Within this address space, you can load any memory segment address between 0000H and FFFFH into one of the four segment registers. Now, suppose that you enter the resident portion of COMMAND.COM, then delete some variables in the BIOS variable range, or change a few entries in the interrupt vector table. The computer will promptly reset.

If you do the same thing in Protected mode, or even load the wrong segment address, the computer reacts differently. Instead of a crash or reset, an exception occurs (more on this later). The operating system maintains control of program execution and the offending program ends.

This protection is possible because, even though Protected mode segment addresses contain 16 bits, additional segment addresses aren't created. Instead, segment selectors are used. This results in an index pointing to a segment table describing the various memory segments. This table is similar to a large array of segment descriptors that tells the processor the base address of the segment. The offset address, indicated by the FAR pointer, is added to this address to obtain the address of the memory location in question. Unlike the segment address in Real mode, this base address isn't multiplied. It's possible for memory segments in Protected mode to begin with the memory address desired, not merely at addresses divisible by 16.

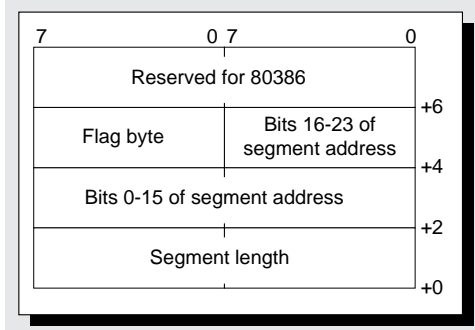
The descriptor table in Protected mode provides a link between a virtual address (segment selector:offset) and a linear address (base address comprising the segment descriptor + offset). In the 80286, the linear address and physical address are identical (i.e., the actual memory location to which access is ultimately gained). The 80386 and 80486 addressing is coded to permit the physical address to be determined initially by this new transformation. We'll discuss this in greater detail later.

Memory access in Protected mode through segment selector and offset



The above description doesn't show the structure of the segment descriptor within the descriptor table. There is more information stored here than just the starting address of the segment in question. The following chart shows the exact structure of a segment descriptor in Protected mode. A

Structure of a segment descriptor



segment descriptor consists of eight bytes distributed among various fields. The first byte is the segment length. Unlike Real mode, the Protected mode segments don't have to be exactly 64K. The first field in the segment descriptor specifies the length of the segment (between one byte and 64K). The next three bytes yield the starting address of the segment in memory. The use of three bytes expands the address width from the 20 bits of Real mode to 24 bits. This also enlarges the physically addressable memory from 1 megabyte to 16 megabytes.

The fact the location of a segment is coded in the segment descriptor instead of in the FAR pointers, which provides access to this segment, helps implement an efficient memory management system. The parallel execution of multiple programs is characterized by the continual allocation and release of memory ranges. This causes memory fragmentation. Constantly shifting these memory segments minimizes this fragmentation.

In Real mode, all references to such a segment within the appropriate program must be adjusted. However, in Protected mode, the segment must be displaced and the segment descriptor must be redirected to the new base address. Now it's possible to continue to use the original segment selector in the various references to the segment, even though the segment now resides at an entirely different location in memory.

Following the base address of the segment, within the segment descriptor, is the byte with the various flags that we'll discuss shortly. The last field contains a word reserved for the 80386 and its successors. This field always contains the value 0 for the 80286. This will also be discussed in more detail later in Section 33.1.3 (Mode Programming of the 80386 and 80486).

Various segment types

Protected mode recognizes three different segment types: data segments, code segments, and system segments. Although data segments can be described and read, program code cannot be executed here. However, code segments can be executed, but

cannot be read or described. The third type, system segments, describes different types of segments. All these segments apply to Protected mode.

Besides the segment type, additional segment attributes are recorded in the flag byte. The meanings of these attributes depend partially on the segment type. A special bit determines whether the contents of a code segment may at least be read. In a data segment, the corresponding bit can be used to block write access to this segment, which makes the block's contents "read only".

Regardless of the type of the segment, the flag byte contains a presence bit that helps program virtual memory management. Whenever the operating system moves a segment to disk, it must set this bit to 0 to signal an error to the processor when it next accesses a memory location in this segment. An operating system routine reloads the segment into memory before program execution continues.

The access bit is also important. The processor sets this bit to 1 on each segment access. When virtual memory management must decide which segments should be removed from memory because additional room is needed, preference can be given to those segments not recently accessed (those with their access bits set to 0).

Privilege levels

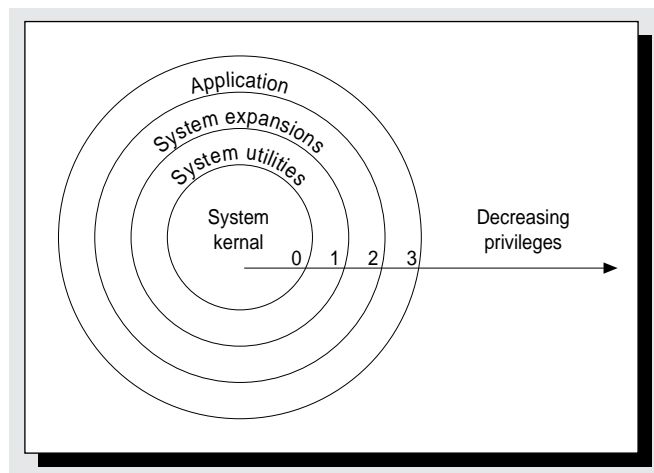
The processor in Protected mode recognizes four different privilege levels for separate programs. These privilege levels specify the execution of various assembly language instructions, and govern access to memory segments. They are designated with numbers 0 through 3 (privilege level 0 is highest, 3 is lowest).

If you only had to distinguish between applications and the operating system, two privilege levels would obviously be sufficient. However, an operating system is usually divided into components that contain different privileges. The highest privileges (level 0) are used by the operating system kernel, which watches over memory management and task switching.

Privilege level 1 is granted to the various operating system services called by programs and operating system utilities. These include file management functions, routines for screen output, and printer control utilities. Privilege level 2 is reserved for operating system extensions that rely on the system services of privilege level 1. In the OS/2 operating system, these include the SQL server and the LAN manager.

Privilege level 3 is for various programs that run under operating system control. Since these programs execute on a lower privilege level than the operating system, they can be controlled by the operating system (but not vice versa). Privilege level arrangement can be viewed as a series of concentric circles, as presented in the following illustration. The innermost circle, the heart of the system, is the kernel of the operating system. As they proceed outward from privilege 0, the privileges diminish as they approach program and user level.

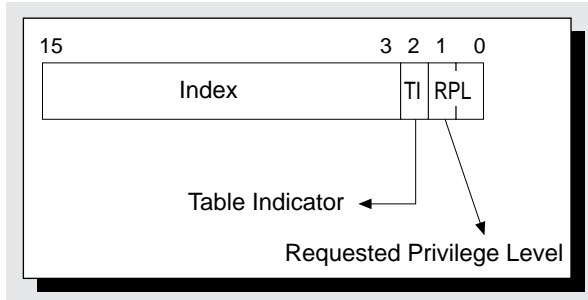
*The four
privilege levels
in the 80286
and successors*



The various privilege levels are important for accessing memory in other segments and transferring program execution to other segments. A program must be prevented from accessing the memory segment of the operating system kernel, or from transferring executions to just any location within another segment without the actual specification of a destination point in a specific routine.

So, the processor compares the privilege levels before accessing memory and signals an error whenever a task tries to access a segment of higher priority. The segment selectors and the segment descriptors contain information on their respective privilege levels.

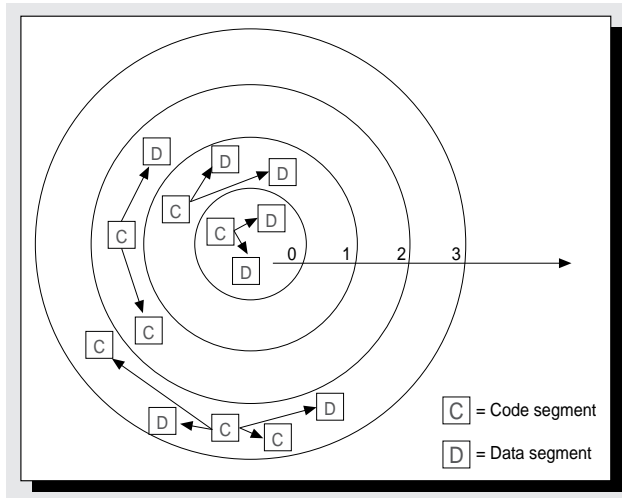
Segment selector structure in Protected Mode



As you can see in the illustration above, the Requested Privilege Level (RPL) is recorded in the low order two bits of each segment selector. The processor also uses two other privilege information sources, the Current Privilege Level (CPL) and the Descriptor Privilege Level (DPL). The privilege entries do the following:

Entry	Description
DPL	The privilege level of a segment, which is recorded in bits 5 and 6 in the flag byte of a segment descriptor.
CPL	The privilege level of the current task. It's determined from the segment descriptor describing the current code segment. The selector of this segment descriptor is found in the CS register.
RPL	The privilege level that is recorded within a selector. It always corresponds to the privilege level of the segment to which the selector points. Hence, RPL = DPL of the segment addressed.

Segment selector structure in Protected Mode



Data and code accesses (jump instructions and subroutine calls) are permissible if:

$$CPL = DPL$$

In other words, the current code segment is on the same privilege level as the code or data segment addressed. If the two privilege levels happen to be different, other rules will apply, depending on the type of access that will be used by the processor as the basis for determining the validity of an access attempt.

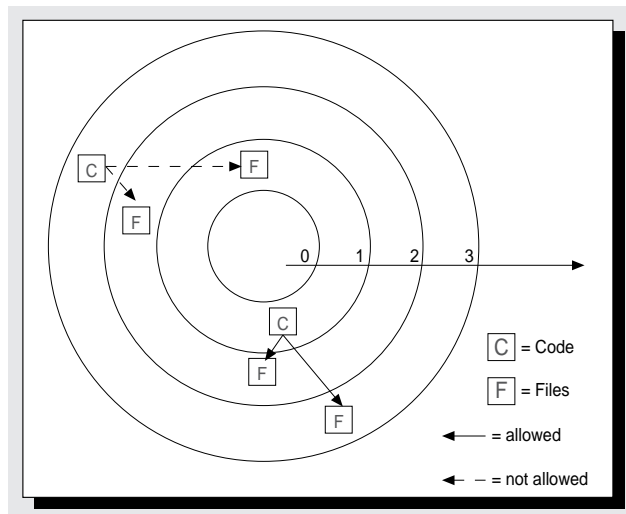
Data access

For data access, this simple rule applies:

$$CPL \leq DPL,$$

This means that a task cannot access a data segment on a higher level than itself. It's impossible for an application to access the data segments of the operating system, but the operating system can send data to the application in a buffer. This rule applies when a selector is loaded into a data segment in one of the segment registers (DS or ES). If one of the protection rules is violated, the processor stops program execution by returning an exception and calls an operating system routine that handles this error.

Data accesses which are allowed or not allowed in Protected mode



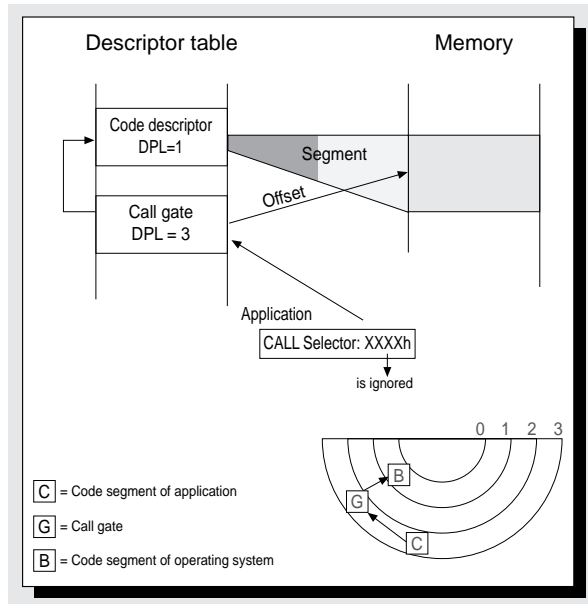
Code accesses and gates

The processor is even more restrictive with code accesses, which is the transfer of program execution by a JMP or CALL instruction. These are only permitted if the segment to which the jump is made is on the same privilege level as the caller. This prevents uncontrolled calls to code on another privilege level. You may be wondering how an application can invoke operating system services, which are on a higher privilege level, while it's on a lower privilege level.

The gates created especially for this purpose provides a solution to this problem. These are special segment descriptors (system descriptors) that occupy eight bytes like any other descriptor and are located in the descriptor table. But, unlike code and data segment descriptors, they don't define a memory segment. Instead, they define a point of entry into a routine whose code segment can be on a higher privilege level than that of the caller.

This segment is defined in the form of an entirely normal selector within the CALL gate descriptor, which must be an executable code segment. Not only the code segment in question is recorded here, but also the offset (i.e., the point of entry into the desired routine). This makes it impossible to jump to just any address within the code segment. Otherwise, it would be extremely easy to drop into the middle of a routine or an assembler instruction.

Using a CALL gate to call an operating system routine



Because the offset address is recorded in the call gate, it loses its significance within the actual CALL instruction and is ignored. Only the selector for access to the call gate is used. However, you must still follow the privilege rules. In this case, the CALL gate is treated like a data segment and can only be addressed if its DPL carries the same or a lower (numerically higher) level than the caller ($CPL \leq DPL$). So, the operating system must position its CALL gates at level 3 if its use throughout all privilege levels is also to be permitted to applications.

An operating system provides CALL gates to each program so they can call various system functions. The contents of these CALL gates (the address of the routine called) is unknown to the program, although its segment selector is supplied by the operating system. Another interesting feature is the conforming segments. In this case, we're concerned with normal coding segments, which are characterized as conforming segments by a special bit in the flag register of its segment descriptor. A characteristic feature of this code segment is the accommodation of its privilege level to that of its caller.

For example, if an application calls an operating system function, accommodated in a conforming segment, the program code contained in the latter is automatically executed at the level of the application (3) instead of at the privilege level of the operating system (0). This makes sense in various situations where the operating function must be viewed as an extension of the application and should not enjoy higher privileges.

Local and global descriptor tables

So far we've been considering only those descriptor tables in which the processor stores the descriptor of the appropriate segment. It's possible in Protected mode to set up as many as 8193 descriptor tables: one global and 8192 local. However, only two descriptor tables are active at a time: the global and a local. Each task is assigned its own local descriptor table. When tasks are switched, a change is made to the descriptor table of the new task. The two descriptor tables are assigned the names global descriptor table (GDT) and local descriptor table (LDT). They are the registers that expanded the 80286 over the 8086.

It's only possible to load these registers from the operating system using the LGDT (Load global descriptor table) and LLDT (Load local descriptor table) instructions. Whenever a task calls these instructions, which do not carry privilege 0, an exception, which automatically stops execution, is issued and a routine is called in the system kernel.

Execution of the LGDT instruction loads the physical address of the global descriptor table and its length into the GDTR register. The length entry enables the processor to determine the number of descriptors in this table ($\text{length} / 8$) and is able

to rapidly recognize an attempt to access nonexistent descriptors. If the global descriptor table contains only 34 entries, each selector, whose index component exceeds the value 33, is invalid because the descriptors are numbered starting with 0.

The TI bit determines whether a selector addresses the global or the local descriptor table. If it contains a 0, the global descriptor is referenced; a one refers to the local descriptor. An application will usually access its local descriptor table where the operating system stores the descriptors of all code and data segments of the task in question. However, the CALL gates for calling the operating system functions aren't located here. Storing information about the CALL gates, which can sometimes number into the hundreds, in each local description table would simply take up too much memory.

The global descriptor table contains not only the CALL gates of operating system functions, but also the code and data segment of the operating system, as well as various other system descriptors. Included among these are the descriptors of the memory segments containing the local descriptor tables. Unlike the GDTR register, which contains the physical address of the global descriptor table, the LDTR register contains only one selector that must point to an LDT descriptor in the LDT descriptor table. So, it's very easy to install a new local descriptor table when switching tasks. It's only necessary to load the LDTR register with the selector of the new local descriptor table.

Virtual address space size

When calculating the size of the virtual address space:Size of, it's necessary to multiply the maximum number of descriptors by the maximum segment size of 64K. To do this, first you must know the number of descriptors that can fit in the global descriptor table and in the various local descriptor tables. For both tables the size is 8192. There are two reasons for this.

First, local and global descriptor tables affect segments. In the 80286, segments cannot be larger than 64K. It's impossible to store more than 8192 descriptors in 64K because each descriptor occupies 8 bytes. Also, you cannot address more than 8192 descriptors in the global or local descriptor table because the descriptor number within a selector is coded with only 13 bits. So, only numbers between 0 and 8192 can be accessed.

For example, suppose the global descriptor table contains 8192 descriptors of local descriptor tables, each of them containing in turn 8192 descriptors of 64K code or data segments. The virtual memory capacity would be:

$$8192 * 8192 * 64K = 1 \text{ gigabyte}$$

which can be addressed (theoretically) only by an efficient virtual memory management system.

Accessing a descriptor

Although the linkage between selectors and descriptors and between the global and local descriptor tables may be complicated, you'll rarely encounter it in application programming. Simply avoid conflict between the selectors.

Essentially, the values used by an application come directly or indirectly from the operating system. An application cannot directly access the local descriptor table or the global descriptor table. Also, it cannot expand this table with new descriptors, or alter existing descriptors. This is solely the task of the operating system kernel, running at privilege level 0. The various instructions needed to manipulate this table can be found at this privilege level.

The operating system also supplies the local descriptor table with the various segment descriptors it needs to receive program code and data when an application is being loaded. 64K segments aren't set up automatically. Only that amount of memory, which is actually required by the segment in question, is released. If the application, because of an application error, seeks access to data or program code beyond the segment end, execution of the program is immediately interrupted by an exception, and an operating system routine is called to handle the error.

Even with the establishment of the various segments and the associated descriptors, reference must be made to these segments within the program, as shown by the following sequence of instructions from an assembler program:

```
MOV AX, segment data
MOV DS, AX
```

In a C program, to transfer a FAR pointer of a function in code segment to a different function, the desired segment selector must be available. In this case, we're concerned with the selector of a code segment, instead of a data segment. The situation in both cases is the same. In this respect, a multitasking operating system functions almost the same as DOS: The head of the EXE file contains the addresses of the instructions within the program code, or of the variables from the constant data segment within which the various program and data segments are referenced. This table allows the loading function of the operating system to write the selectors of the addressed segments directly into the program code or the variables in question. Operating system calls will probably be handled in the same way, except that a CALL gate selector is required.

Some operating system functions return selectors as function results, especially the functions from the memory management area. These functions can be treated as a component part of a FAR pointer, like normal segment addresses. You can avoid manipulating this presumed segment address, which is frequently required in DOS programming. Manipulating this presumed address could create a selector pointing to an entirely different, or even nonexistent, descriptor.

The RPL field is important to the interaction between the operating system and application selectors. The preceding section indicated the privilege level of a task is always recorded in the RPL field of a selector by the task in possession of this selector (RPL = CPL).

It's possible to load any value into this field. However, when assigning selectors, the operating system will always enter the privilege level of the respective task into the RPL field of the selector transferred. If such a selector is later transferred to an operating system function, permitting the latter to access a buffer, the high privilege level of the function shouldn't be used. Instead, this access should be executed at the privilege level of the caller, which prevents the latter from accessing data levels on all four privilege levels via the operating system function.

So, in these instances, the RPL field in the selector is set for the privilege check and points to the privilege level of the caller instead of to the privilege level of the operating system.

Shadow registers

When the various memory segments are addressed using selectors as pointers to segment descriptors (instead of their physical addresses), you can no longer address physical addresses in the segment register, as we demonstrated earlier. The processor must address the respective descriptor in the global or local descriptor table using a segment register selector, and from that determine the position and length of the segment in question. If the processor tried to do this in the case of each instruction, in whose execution one of the segment registers is involved, too much time would be wasted.

The data on the physical address of a segment and its length are loaded, when a segment register is loaded, into a shadow register. A shadow register exists for each segment register. Besides the visible 16-bit selector, each of the four segment register contains an additional invisible part that is 48 bits wide. This applies to the GDTR, LDTR, and IDTR registers, but these shadow registers manage different data.

Loading the shadow registers and the associated validity and privilege checks account for slower execution of all instructions affecting the content of the segment registers. For example, only two clock cycles are needed to execute the following instruction in Real mode:

```
MOV DS, AX
```

The same instruction line in Protected mode requires 18 clock cycles. This doesn't significantly affect execution speed when these instructions are rarely encountered within an application. However, this usually isn't the case, especially with high level language programs that are compiled using memory models with FAR pointers. For example, in C all memory models except SMALL and TINY use FAR pointers. The restriction of memory segments to 64K frequently makes any other procedure in Real mode impossible.

Aliasing

Data cannot be written into a code segment and the content of a data segment cannot be read by two tasks, whose code segments have different privileges. These are two of the many protection rules designed to ensure that a multitasking operating system runs smoothly.

However, there are other situations, in which these protection rules don't apply. For example, how can the operating system load the program code into an application segment, in which its contents may not be written? How can two tasks on different privilege levels gain access to the same data segment? In these situations, a mechanism known as aliasing circumvents the protection rules of the processor.

Although the processor controls access to a memory segment through its descriptor, it cannot prevent several descriptors from being entered in one or more different descriptor tables for the same memory range. So, it's possible that a memory range, which is already identified by a descriptor as a code segment, will be described partially or entirely as a data segment. The second descriptor is called an alias because it permits addressing a range in memory under another "name."

This makes it possible not only to load program code into a data segment, but to solve the problem of common memory usage. You can then set data segment descriptors in the task's LDT to duplicate the privilege level of the task. This duplicate then accesses the data segment using the descriptor.

Aliasing is useful whenever you must bypass the protection mechanisms of the processor. However, it can also cause several problems. So, the operating system uses aliases only in very special situations.

Hardware access

You can also restrict hardware access by using I/O ports. The INS and OUTS instructions (80286 extensions of the IN and OUT instructions) can be privileged.

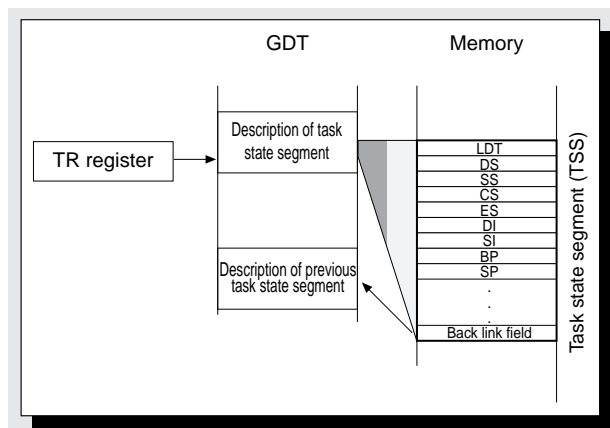
The IOPL (I/O Privilege Level) bits in the flag register are responsible for this privileging. They indicate the privilege level a task must have before the task can execute the named assembler instructions. If both of these bits contain the value 1, only tasks on the 0 and 1 levels of privilege may execute this instruction. If they are executed from levels 2 or 3, the processor launches an exception, which activates an operating system routine. This mechanism is only useful if the right to alter the contents of the IOPL bits isn't limited to the various applications. Conceding these rights could give applications the power to take over I/O execution themselves. Changing the flag register isn't difficult; simply push the desired contents onto the stack and execute the POPF instruction.

The 80286 developers made slight changes to the way the POPF instruction works. Although the POPF instruction can still be used at all privilege levels for setting the various flags in the flag register, this doesn't apply to the two IOPL bits. POPF changes the IOPL bits only when the calling task is executed on privilege level 0. This prevents normal applications from influencing the IOPL bits.

Task switching

The task register (TR) and the task state segments (TSS) support rapid task switching in Protected mode. These registers control memory ranges and comprise 44 bytes, which store the contents of all processor registers when a task is interrupted.

Linking the task register to a task state segment



Also, a pointer to the task state segment of the previously active task is stored. This enables the processor to easily return to this task. As with all memory segments, each task state segment possesses a descriptor, which must be accommodated in the global descriptor table. However, this is a special system descriptor that points to the location and size of its segment like a normal descriptor. This descriptor is referenced by the TR register, which contains the selector for the descriptor of the current task. In the case of a task switch, the contents of the processor registers can be stored immediately in its task state segment.

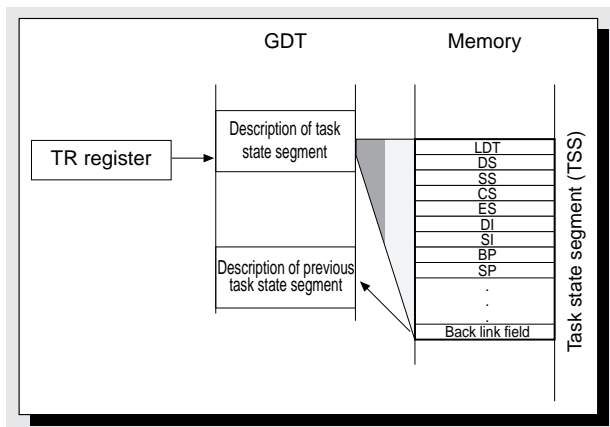
A task switch can occur in several ways; usually a FAR JMP or FAR CALL instruction is used. In this type of instruction, an offset address must be indicated in addition to a selector. However, only the selector is important for the task switch, which is similar to a function call by using CALL gate. The offset address is essentially ignored. This selector can be either the selector of a TSS descriptor from the GDT or the selector of a task gate. Similar to the CALL gate already described, such a gate serves as an intermediary and represents a system descriptor that can be accommodated either in the global or one of the local descriptor tables. The only significant data it contains is the selector of the TSS descriptor from the GDT, which identifies the new task.

Following the task gate route, the processor arrives at the required descriptor of the task state segment from the global descriptor table. The task change can be completed with its help; the former contents of the TR register are processed first (i.e., the pointer to the descriptor of the current task state segment). This segment initially contains the current contents of the processor registers. The TR register is loaded with the TSS selector of the new task. In any case, the former content of the TR register is only stored briefly to permit it to be entered in the back link field of the new task state segment.

The processor contents stored here are loaded from the new task state segment. Because the CS and IP registers are also included among the secured registers, program execution continues at the exact location where the new task was interrupted.

If the operating system wants to set and start a new task, it will first set up a TSS descriptor in the GDT and use it to establish a task state segment for this task. Because the task wasn't active yet, the memory locations for the processor registers inside the new TSS must be initialized manually. The segment registers, the stack pointer, and the IP register are especially important. The starting address of the new task is established by CS:IP register pair. Program execution is then transferred to the new task by means of a FAR JMP with the selector of the TSS.

*Task switching
with a FAR JMP
and task gate
descriptor*



Interrupts and exceptions

Interrupts are handled differently in Protected mode than in Real mode. The processor still recognizes 256 different interrupts, and still interacts between an interrupt and its interrupt handler but the Real mode interrupt vector table no longer applies. The solution is the Interrupt Descriptor Table (IDT), which is structured like a global or local descriptor table. Its starting address is stored in physical memory, and its length in the IDTR processor register. Access to this register in Protected mode is reserved for the operating system kernel, and this access executes on privilege level 0.

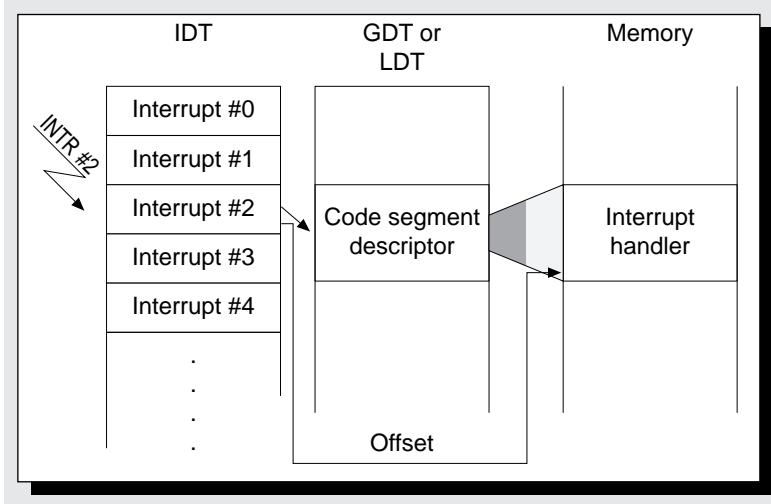
Unlike Real mode, an application isn't able to influence interrupt execution. This is an important prerequisite for a stable multitasking system. Under DOS, many TSR problems are caused as a result of the free accessibility to the interrupt vector

table. If an interrupt is triggered over the INTR processor line by an external device, or as the result of an INT assembler instruction, the processor enters the interrupt vector as an index into the interrupt descriptor table and reads the corresponding descriptor. This can be either an interrupt or a trap gate, since their structures are very similar. Both contain a selector and an offset address.

The selector serves as the access key to a code segment descriptor in the global or local descriptor table. The processor notifies the code segment containing the interrupt handler for the triggered interrupt. The offset, however, serves as the point of entry into this code segment, thus representing the start address of the interrupt handler within this code segment.

The difference between an interrupt and a trap gate consists of the association with the interrupt flag in the interrupt register. In Real mode, further interrupts usually cannot be called immediately after calling one interrupt, because the processor automatically sets this flag to 0 until returning to the interrupted program. The processor proceeds in much the same way with an interrupt call through an interrupt gate. However, if it encounters a trap gate, the interrupt flag in the flag register isn't deleted. This makes it possible to trigger additional interrupts even during execution of the interrupt handler.

Using the interrupt descriptor table (IDT) to call an interrupt in Protected mode



Exceptions are essentially interrupts, except they are directly called by the processor if an error occurs during the execution of an instruction. A reason for this could be infringement of privilege rights during access to memory segments or the specification of addresses following the end of the segment in question.

The processor recognizes interrupt requests from the interrupt controller, but only between the execution of two instructions. So, exceptions are triggered in the middle of the execution of a instruction. Once the error has been handled and removed by the exception handler of the operating system, the execution of the program must begin anew. The exceptions of the 80286 control occupy interrupts 0H through 10H. They are shown in the following table. Depending on the exception, the processor pushes data onto the stack before calling the exception handler, which more closely specifies the cause of the exception.

Exceptions in Protected Mode					
No.	Meaning	No.	Meaning	No.	Meaning
0	Division error	1	Single step	2	NMI (memory error)
3	Break point	4	Break caused by INTO	5	Break caused by BOUND
6	Unknown command code	7	Coprocessor not available	8	Double error
9	Coprocessor segment overflow	10	Invalid Task Status Segment	11	Segment not available
12	Stack error	13	Invalid segment access	16	Coprocessor error

By programming for the Protected mode of the 80286 or one of its successors, you leave the world of 8086 compatibility. Here's a short list of the most important new instructions. These instructions may also be used in Real mode.

Op code	Definition
BOUND	Checks whether the contents of a register is within a specified range.
ENTER	Creates the stack frame and temporary storage required by many high level languages by copying function parameters onto the stack and reserving memory for local variables.
INS	Transfers data from a specified I/O port into a memory operand pointed to by the ES segment register and DI/EDI (the destination index register) and updates the index to prepare for the next transfer.
LEAVE	Counterpart to ENTER, but reverses the action of the ENTER instruction. Deletes the transferred function parameters and the local variables from the stack.
OUTS	Transfers data from a memory operand pointed to by the source index register to the specified I/O port and updates the index to prepare for the next transfer.
PUSHA	Saves the 16-bit or 32-bit general registers on the stack.
POPA	Restores registers from the stack.

The instructions from the following table aren't suitable for Protected mode programming because you must provide your own operating system services. All these instructions are privileged and may only be executed on privilege level 0. This makes them accessible in Real mode and are treated by the processor like a task with the privilege level 0.

Op code	Definition
ARPL	Checks and tests the privilege level of a code segment.
LAR	Loads the access rights byte of a descriptor.
LGDT	Loads the address and the length of the global descriptor table in the GDT register.
LIDT	Loads the address and the length of the interrupt descriptor into the IDT register.
LLDT	Loads the local descriptor table register.
LMSW	Loads the machine status word from the source operand.
LSL	Load segment limit.
LTR	Loads the task register (TR).
SGDT	Stores the contents of the GDT register.
SIDT	Stores the contents of the IDT register.
SLDT	Stores the contents of the LDT register.
SMSW	Stores the machine status word.
STR	Stores the contents of the task register TR.
VERR	Verifies a segment for reading.
VERW	Verifies a segment for writing.

Protected mode on the 80386 and 80486

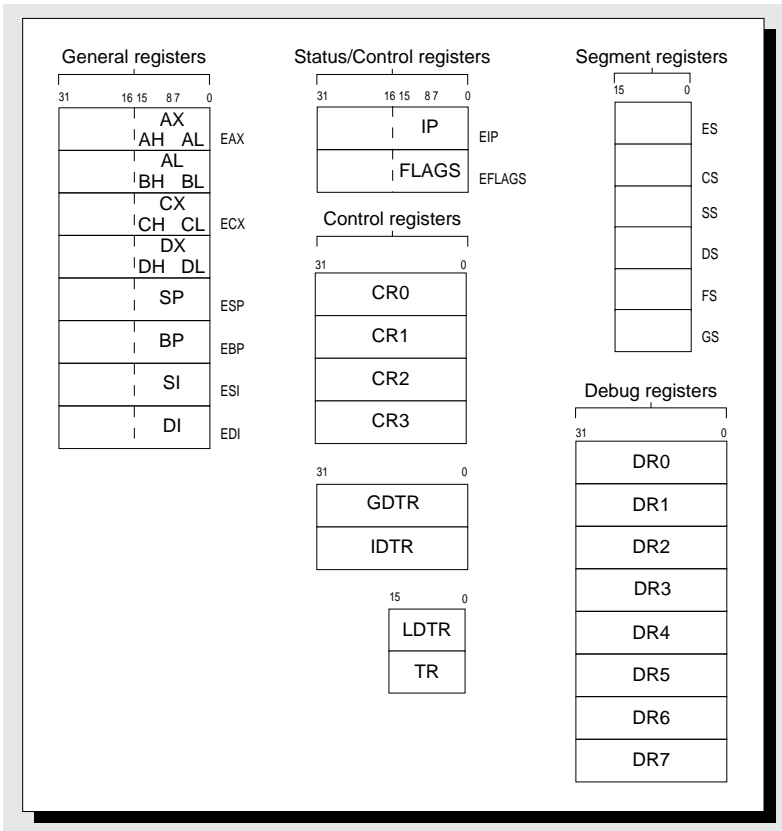
Protected mode operation on the 80386 and 80486 processors changes only slightly from that of the 80286. However, a few new concepts were included that shouldn't be used by a multitasking operating system. These include the paging of 4K memory blocks (vital to virtual memory management) and locking out individual I/O ports. These extensions are discussed in the following sections.

The Real mode changes are more important in these processors, especially the availability of 32-bit registers and the changes to the instruction set to accommodate the new register width. We'll start with an overview of the 80386 and 80486's register complement, and how it affects Protected mode.

The 80386 and 80486 registers

The introduction of the 80386 marked the start of the 32-bit processor age. This is evident in the processor's register structure, since most registers have a 32-bit width. The AX, BX, CX, DX, DI, SI, and BP registers are still six bits wide. However, these 16 bits form part of an expanded 32-bit register, representing the low word of the 32-bit register. These 32-bit registers are named EAX, EBX, ECX, etc. The E represents extended.

The register complement of the 80386 and i486 (no floating-point registers)



The 8-bit registers (AH, AL, BH, BL, etc.) are available but the upper 16 bits of the E registers cannot be divided into two 8-bit registers. The number of available 8-bit registers remains unchanged, maintaining downward compatibility with the earlier 80xxx chips. Instructions, such as MOV, SHL, ADD, etc., apply to 8-bit, 16-bit, and 32-bit registers alike. The 80386 and its successors use this to achieve higher speed, especially when processing DWORDS (long in C and LongInt in Pascal).

The illustration on the previous page shows both wider registers than the 80286 and new registers. Notice the FS and GS registers. These two new segment registers can be used for memory access just like the ES register. As before, DS still acts as the default register for accessing data, and CS acts as the default register for program execution.

Like the 80286, the various segment registers in 80386 Real mode mirror the segment addresses of the segments addressed. While in Protected mode, they accept selectors for segment descriptor access from the local or global descriptor table. The GDTR, LDTR, IDTR, and TR registers, which are used for memory management and task control, also remain the same. These registers can only access the operating system kernel on privilege level 0. Four 32-bit control registers have been added: CR0, CR1, CR2 and CR3. These control registers are reserved for the operating system kernel (more on this later).

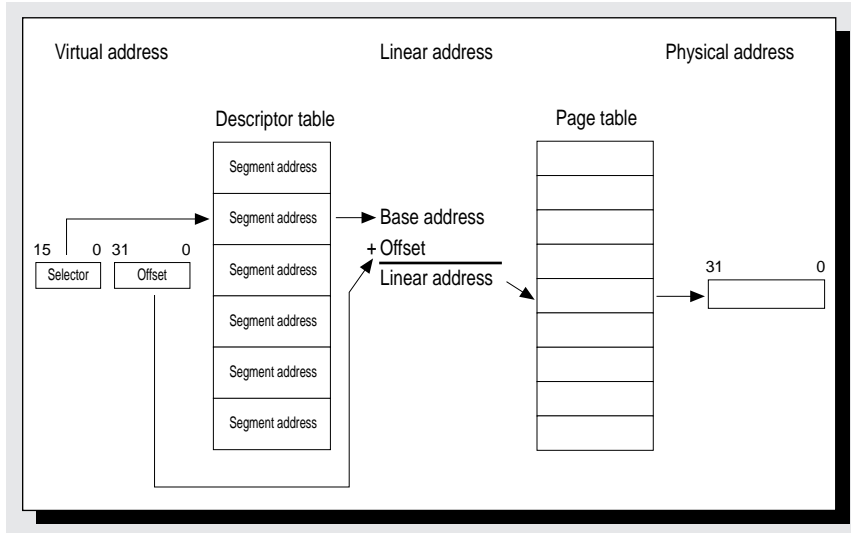
The 80386 includes eight debug registers called DR0 through DR7. These registers allow you to set four different break points that can be triggered at any point in a program (e.g., when execution reaches a specific instruction, when a read access occurs,

This is the best way to implement virtual memory management. The memory blocks to be moved in and out are always various lengths. They are always complete segments that (in the 80286) can have a length between 1 byte and 64K. However, only part of the segment may be needed. This segment can be removed in the next instant when another task is executed. In each case, the entire segment must be unloaded, then reloaded. Also, the size of the unloading file (or swap file) steadily increases.

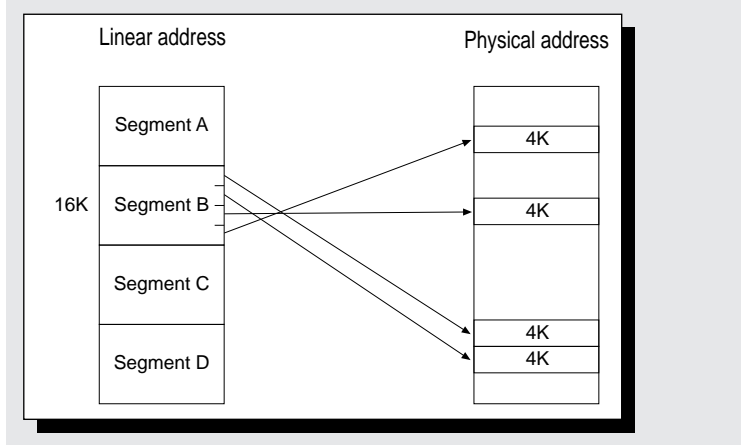
Now, imagine that you want the system to unload segment B. However, there's no space left in the swap file because of segment A, which was just reloaded into memory. Fragmentation also occurs in RAM because the various loaded and unloaded segments are different sizes. The virtual memory management continually compresses the memory segments. This occurs transparently, without interfering with the various programs (the segment descriptor changes, instead of the segment selector). This process is still time-consuming.

These problems are solved if the sizes of the blocks to be loaded and unloaded are kept constant, and this process is inserted after the segmentation process. This places it on a lower level and in makes it transparent. The 80386 and its successors implement a paging mechanism to solve this problem. The paging is based on a group of 4K memory blocks. When this mechanism is disabled, the linear addresses resulting from the dissolution of segment selectors and offset addresses represent physical addresses, just as in the 80286, and these addresses can actually be accessed by the processor.

Physical address formation with paging enabled



When a segment appears in the linear address space, different addresses can occur



Once paging is enabled, the *page table* models the linear addresses after physical addresses (i.e., the linear addresses are no longer identical to physical addresses). If we view this in executing a single memory access instruction, this slows the execution speed. However, if we think of this as part of a multitasking operating system, the mechanism is very efficient for implementing a virtual memory management system. Using constant memory blocks does more than simply manage the swap file. It also prevents the swap file from continually increasing. Memory can be allocated in 4K pages, instead of the entire 1 Gigabyte segment.

Memory compression is also no longer needed, because the blocks can be merged into any linear memory address by using the page table. The four blocks of a 16K memory segment can be stored at entirely different physical addresses to form a continuous memory block within the linear address space, and can be addressed by an application.

What appears in the linear address space to be a continuous segment, can occupy entirely different addresses in the physical address space.

Page tables

The PG bit in control register 0 (CR0) is responsible for making the paging mechanism active. This bit is set to 0 after the system is started in Real mode. Linear addresses are modeled directly on physical addresses. Therefore, paging doesn't occur. However, once Windows 95 switches the processor to Protected mode, it sets this bit to 1 to make the paging mechanism active. Each memory address in Protected mode the processor processes within the frame of a machine language command is assigned to a page. Memory access is then redirected to this page. As a result, the address visible to a software program is completely detached from the actual storage location of the data in memory.

The size of each page is 4K. Each page begins at a physical address divisible by 4K. The 4 Gigabyte (232) linear address space of the 80386 and its successors is divided into 220 pages, each containing 212 bytes (4K). The calculation is correct because 220×212 again yields 232.

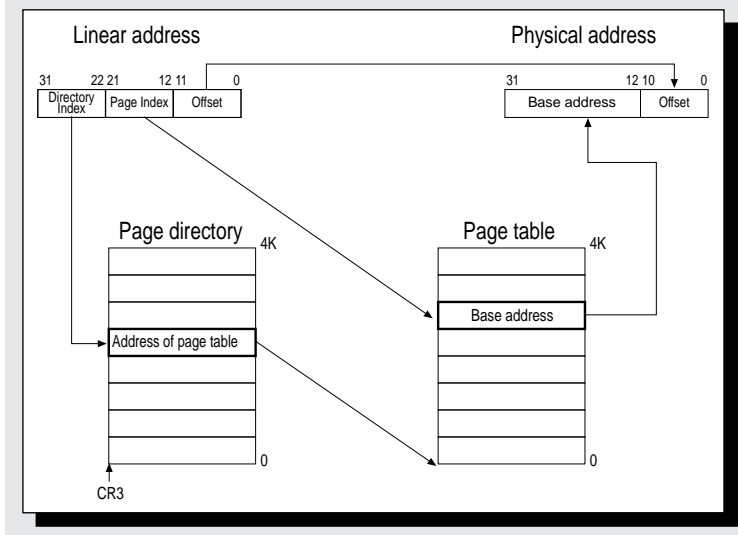
The specification of 4K limits makes it possible for the lower 12 bits of a linear address to be incorporated directly into the physical address. They represent a type of offset in the respective page. The upper 20 bits of the linear address indicates the number of the page containing the memory location addressed. They're considered to be an index in the page table. The physical base index of the respective page can be taken from this page table.

The entries in the page table are respectively 32 bits wide. Only 20 bits are needed for the base address. It must begin at a physical memory location divisible by 4K. The lower 12 bits is 0 in each case. The lower 12 bits contain various flags for the virtual memory management system. For example, the flags indicate whether the page is currently present in memory or must first be loaded from the swap file. Only the upper 20 bits from the page table entry are used to form the physical address. They're complemented by the 12 lower bits of the linear address, which are used unchanged. That gives you the 32 bit wide address required for accessing memory and redirecting each memory access to the proper page.

Unfortunately, the mechanism with the page table has a catch: It uses too much memory. The page table, with its 220 entries of 4 bytes each uses up 4 megabytes. Sometimes, this amount would represent all the available RAM memory for a user.

That's why the developers at Intel refined the mechanism in a critical area. Instead of one large page table for the entire 32 bit address space, they chose a two-stage organization with multiple page tables. While this saves much disk space, it nevertheless puts the processor in the position to clearly map every virtual address. The page directory handles several smaller page tables. As the following illustration shows, the upper 10 bits of the linear address constitute the index in the page directory, which consists of 1024 entries. Since each entry contains 32 bits, this table uses 4K of memory. The address of the page directory is set through the CR3 register, which plays a key role for correct operation of the system. If this register is given the wrong setting, you can bet the rent money that the system will crash in the next few microseconds. That's why access to CR3 is only open to the system code.

Converting a linear address using the page table structure



The individual entries are pointers to the addresses of the various page tables in the physical address space. This is where the processor first learns the address of a page. The entry number used in the respective page table for calculating the address is produced from bits 12 through 21 of the linear address. It's divided into two parts:

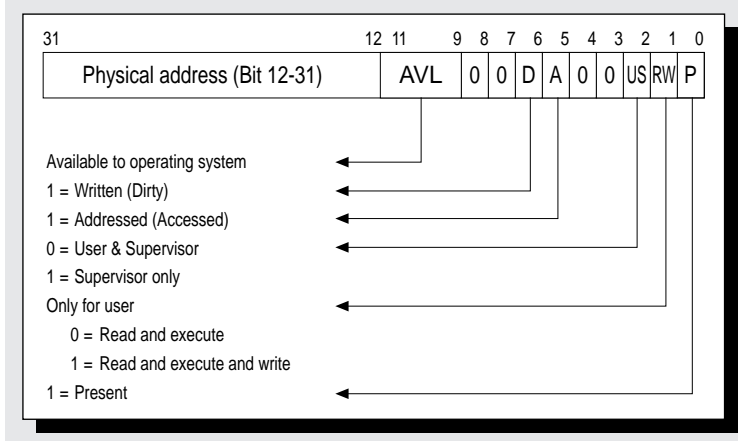
1. 10-bit index for the page directory
2. 10-bit index for the page table in question

The page directory and each page table contains the addresses of 1024 pages and occupies 4K of memory.

Because of the page table structure, each entry within the page directory occupies a 4 megabyte range within the linear address space. The advantage of this is that the system can create the page tables as needed. Only one page table is required if only 4 megabytes were assigned to the caller through virtual memory management. The next 4 require another page table, etc.

As a result, at first only one entry within the page directory is initialized. All other entries in the page directory can initially be marked invalid by setting special flags in the lower 20 bits. Setting these special flags instructs the processor to trigger exceptions (more on this later). Thanks to this procedure, only 4K is needed in the page table for every four megabytes of memory. This is less than the four megabytes in a continuous page table for all the pages in the linear address space.

Structure of a page table entry



Page unloading strategies

The virtual memory management system performs its duties by using various flags in the lower 20 bits of the page table entries. The Present flag (bit zero of the page table entry) must be set to 0 by the operating system if a page was unloaded. The processor then triggers an exception when this page is accessed. This enables the operating system to take control of program execution, loading the page and initializing the page table entry with the new base address of the page in memory. The Present flag must then be reset to 1, or the processor will again cause an exception on the next access.

The Dirty bit and the Accessed bit are also useful in virtual memory management. The Accessed bit is automatically set to 1 by the processor before accessing the page of a page table entry. This flag is even set to 1 if only one of the total of 1024 pages in a directory entry is addressed.

Because this flag is set only to 1 by the processor, and never to 0, the virtual memory management system can use it for marking addressed pages. It must set this bit to 0 after loading, or after establishing a page. The Accessed bit can then be used to test whether this page was already addressed. This information is very helpful when space in physical memory is limited, which making it necessary to shift pages into the swap file. Then it's possible to give priority to recently addressed and immediately needed pages.

The reasons why the pages just addressed should be unloaded are many. Supporters of this theory argue these pages were already in the queue, while the others are only to be addressed in the next instant. A further criterion for this decision is supplied by the Dirty bit, which is set to 1 only when write accesses occur. Reading this bit makes it possible to distinguish altered pages from unaltered pages. Priority should be given to unaltered pages during unloading because they're retained in their original form in the swap file and will have to be reloaded again later. In this way, the processor contributes toward the goal of using only those resources that are necessary for virtual memory management.

Protective mechanisms

The remaining flags in the page table entries constitute a protective mechanism (similar to the one found at the linear address level). Distinctions are made between supervisor and user code instead of among four privilege levels. All tasks carrying privilege levels 0, 1, or 2, which can thus be part of the operating system, have supervisor code authority. Tasks from privilege level 3 are designated as user codes. Under Windows 95 this means that only the actual kernel is considered supervisor code. Application programs, device drivers and system utilities, on the other hand, run as user code.

If a page's user/supervisor flag is set, which indicates a supervisor page, user tasks are generally denied access to this page and create an exception. However, if this flag isn't set, access is open both to user tasks as well as supervisor tasks. The Read/Write flag determines access to user tasks. If this flag contains a 0 value, the page in question may be read and executed, but not written. If it contains a value of 1, the page allows writing. Errors generate exceptions in this case also. This way parts of the operating system can also be inserted into the address space of an application without enabling the user code to access the parts. For example, this could be done for buffers that the system manages itself but maintains for the process in question, placing the buffer in the process' address space. Another example would be system program code. It must appear in the address space of the application so it can be called but cannot be overwritten.

Selective I/O port blocking

The 80386 and its successors have a slightly different approach than the 80286 for determining whether an I/O port may be addressed by a task. Although only the IOPL flag in the flag register and the privilege level of the task in question count in the 80286, it's now possible to set up an I/O permission bitmap in the task state segment of a task. This bitmap is a large bit array with up to 216 entries. Provisions are made for all 65536 possible ports to be tested as a group, to determine whether a basic access without reference to privilege level will be possible, or whether a test via the IOPL flag should occur. The value 0 indicates free access; 1 represents the IOPL test.

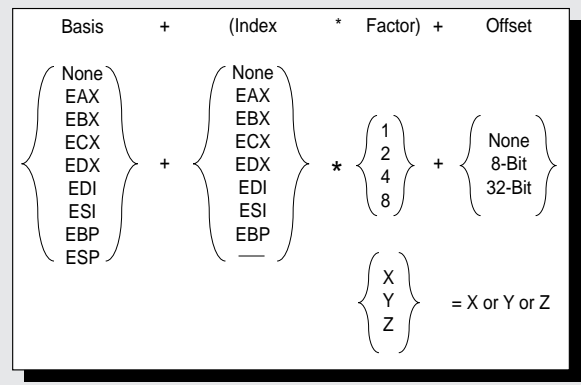
The starting offset within the task page segment isn't fixed, but can be selected by the operating system. It must be specified at offset address 66H in the task state segment, which, in the 80386, has grown to at least 102 bytes because of the expanded processor registers. From this starting address and the length of the task state segment stored in the descriptor, the processor determines the size of the bit array. The ports, which are no longer covered by the bit array, are subject to the IOPL test.

If the offset address of the bit array is larger than or equal to the segment length, there won't be a bit array and the IOPL test is executed. If more room is left between the offset address and the segment length than the maximum length of 8K, all ports will be covered by the array. If none of these conditions are met, the interval between the end of the task state segment and the start of the bit array is calculated and this value is multiplied by 8 to determine the number of ports represented in it.

More flexible addressing modes

The types of addressing were also significantly expanded with the introduction of the 80386. All general registers are viewed as base and index registers, both in Real mode and Protected mode, instead of only certain combinations of the SI, DI, and BP registers. Also, it's possible to multiply an index register by a factor of 2, 4, or 8 to permit rapid access to word, DWORD, or QWORD arrays. A constant 8-bit or 32-bit offset can be added, as the following illustration shows:

*The four
privilege levels
in the 80286
and successors*



The 80386 and 80486 processors include the ability to accommodate data and variables, according to their size in memory, to word, DWORD, or QWORD limits. Otherwise, access will last significantly longer because several read and write accesses are needed to compose the operands.

New instructions

As with the 80286, the 80386 was also extended by a few instructions or operation modes for instructions already familiar to programmers. These changes are for non-privileged instructions. System instructions on the 0 privilege level don't have any changes. The following table summarizes the new options hidden in the 80386 instruction set:

BT,BTC,BTR,BTS	Instructions for testing and setting bits in registers and variables.
BSF,BSR	Instructions for scanning bits in registers and variables.
LSS,LFS,LGS	Loads one of the segment registers: SS, FS, or GS.
Jxx 32-bit	Conditional jump instruction (JC, JA, JBE, etc.) with 32-bit displacement.
MOVSX,MOVZX	MOV instruction with automatic expansion of the prefix or setting the remainder register to 0.
MOV DRx,Reg	Load one of the debug registers.
MOV Reg,DRx	Load a register with one of the debug registers.
MOV CRx,Reg	Load the control register.
MOV Reg,CRx	Loads a registers with the control register.
SHRD,SHLD	Shifts DWORDS left or right.
SETxx	Loads one of the various 8-bit registers with an indicated value, if the condition of the instruction is met. Functions like a combination of CMP, Jxx, and MOV.

Virtual-86 mode

Virtual-86 mode (V86) mode was created as a compromise between Real mode and Protected mode. Many EMS emulators (e.g., Microsoft Corporation's EMM386.SYS) and multitasking environments (e.g., DesqView/386) use V86. When a system operates in V86 mode, the computer is running in Real mode, while retaining the background memory management, task management, and privilege rules found in Protected mode. When you run a program in V86 mode, it runs like an independent V86 task, while the Protected mode mechanisms remain hidden. The program sees only its one megabyte address space, which is addressed according to the standard rules of Real mode:

```
segment_address * 16 + offset_address
```

From the V86 task's point of view, we're concerned with ordinary segment addresses instead of selectors, as in Protected mode. A program can load any value into one of the segment registers without having to worry about an exception occurring. The processor issues an exception for a V86 task if it uses one of the 80386's 32-bit address modes to generate an offset address outside the 64K segment limit. All the 80386 and 80486 address types are available for a V86 task, including the use of 32-bit registers and offsets, with segments larger than 64K as the exception.

V86 mode is used for executing normal DOS programs that rarely access the expanded capabilities of the 80386 and its successors. Only linear addresses exist in V86 mode because of the lack of selectors or descriptor tables. So, the one megabyte address space of a task is patterned after the first megabyte of physical RAM, as long as paging is disabled. If paging is enabled, it's possible for the address space of a V86 task to accommodate itself using the page directory and the necessary page table in any range in physical memory.

While a task is executed in V86 mode, it's possible for other tasks to be active in more than V86 mode. Protected mode is available both in the 80286 compatible 16-bit version and in the 32-bit version of the 80386 and 80486. These modes are usually used by a virtual control monitor, which controls the execution of the DOS program in V86 mode in the background, giving it the feel of a normal DOS machine. In the subsections on EMS emulators and multitaskers, we'll discuss how this occurs.

Protected mode also handles the interrupts and exceptions triggered by a V86 task. The processor switches to Protected mode and the interrupt or exception handlers are called using the associated gate in the interrupt descriptor table. The V86 task isn't responsible for the initialization of this table and preparing the interrupt and exception handlers. This is controlled by the virtual control monitor running the task.

Switching to V86 mode

Several ways are available to switch from Protected mode to the V86 mode with the 80386 and its successors. The most common method is task switching through a task gate. The operating mode, in which the new task is executed, is controlled by the VM flag from the EFLAGS register, which is loaded from the task gate segment of the new task during the task change. If this flag is set, the program runs in V86 mode. If the flag isn't set, normal Protected mode is used.

We described how a new task can be "launched" by generating a task state segment (using an associated task gate and its call). In V86 mode, simply initialize the VM flag within the task state segment with a 1 to begin task execution in V86 mode. V86 tasks always run on the lowest privilege level (3). All instructions that influence the content of the interrupt or VM flags within the EFLAGS register (PUSHF, POPF, INIT, IRET, CLI and STI) are subject to the IOPL test. They may be executed only if a value of 3 is in the IOPL flag. This prevents V86 tasks from acquiring the capacity to switch into Protected mode.

It's also possible to control changes in the interrupt flag. DOS programs have a tendency to delete this flag for a certain time interval to suppress INTR requests. This usually isn't a good practice for a virtual control monitor. During the time the INTR request is suppressed, interrupt requests for other tasks get backed up and cannot be processed quickly enough.

Exceptions are generally triggered from the IOPL for all attempts to access the EFLAGS register. Changes to the addressed flags are prevented. Permanently calling these exception handlers slows down program execution. It's recommended that "safe" DOS programs be supplied with IOPL 3. In V86 mode, the instructions IN, OUT, INS, and OUTS aren't privileged, as they are in Protected mode. Before executing such an instruction, the processor requests the I/O permission bitmap from

the task state segment of the task in question to selectively block or release individual ports. Accessing a blocked port generates an exception and the virtual control monitor gains control of program execution. This gives it the ability to virtualize certain ports. We'll look at this process in more detail in the section on multitaskers.

As you can see, V86 mode supplies the means for executing DOS programs, even in multitasking operations. The following section explains how this is done.

Protected Mode Utilities

The V86 mode, which we described in the previous section, is mainly suitable for developing Protected mode DOS utilities. It's possible to develop a monitor program that runs in Protected mode, but controls one (or more) virtual machines in V86 mode. This provides DOS with an "assistant", which controls all the steps without being noticeable. The following two sections show what happens when V86 mode is combined with DOS.

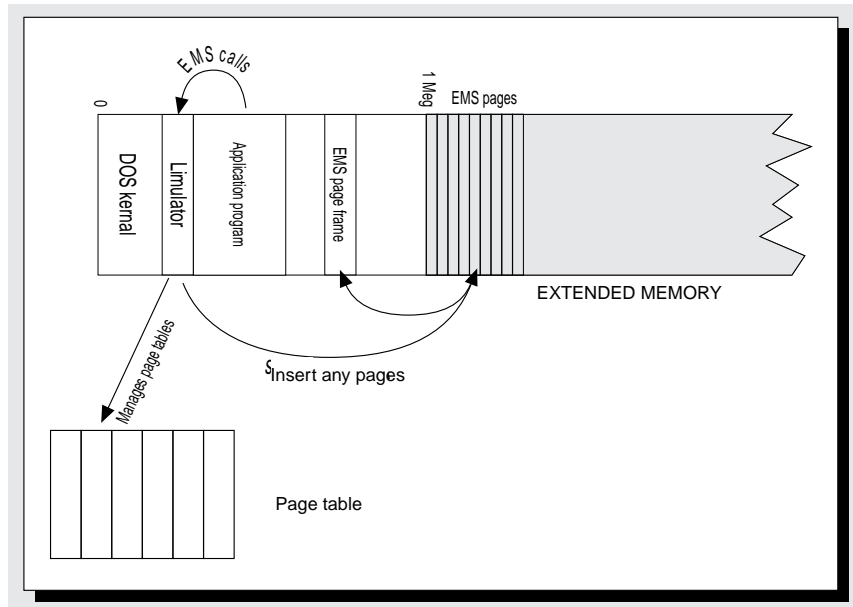
EMS Emulators and memory management programs

EMS (Expanded Memory Specification) simulators are available for systems ranging from the 8086 to the 80486. They're responsible for making expanded memory available according to the LIM (Lotus-Intel-Microsoft) specification. The EMS emulators operate according to different principles, depending on the Intel processor for which they were designed.

The prerequisites for these utilities and their effectiveness have improved with each generation of processor. All of these utilities make the LIM specification available, but often the EMS memory is still inaccessible. Using an EMS emulator is hardly worthwhile for 8086 processors because it uses the available EMS memory on a hard drive file and also unloads there. This process is too time-consuming. The EMS window, with its 64K, must be located in RAM below the 640K limit (conventional memory). This means that you'll lose valuable main memory. EMS emulators for the 8086 aren't very useful.

However, there's a difference when considering the 80286. Although it's possible to take the simulated EMS memory from the hard drive, you shouldn't do this. Instead, you should use the extended memory above the 1 megabyte limit. This memory area is available on almost any AT. The EMS emulator uses this as expanded memory by copying memory from extended memory into the EMS page frame with a special BIOS function.

EMS emulator operation in the 80386 and successors



In any case, you must locate the page frame below the 640K limit. When EMS memory is frequently used, you should install an EMS expansion card, even in 80286 computers. However, many ATs (especially those with the NEAT BIOS chip set) can configure extended memory as expanded memory.

The introduction of the 80386 finally allowed EMS emulators to achieve a significant breakthrough. Various performance characteristics of this processor were designed specifically for EMS emulators. The basis for making expanded memory available is the extended memory above the 1 megabyte limit. However, unlike the 80286, in this case, the memory doesn't have to be copied into a page frame.

The 80386 and its successors provide an efficient paging method. This method makes it easy to model extended memory in the EMS page frame and use the area between 640K and 1 megabyte, even when physical memory doesn't exist there. That's how efficient memory management programs like 386-To-The-Max, QEMM386, and EMM386 perform this task.

However, one small problem remains. In Real mode, from which DOS programs request and use EMS memory, you cannot use 80386 paging. You must use the V86 mode, which means the EMS emulator must be switched into V86 mode at the very start, then continue in the background as the virtual control monitor. We'll discuss this in more detail in the section on multitaskers.

Memory management programs

Memory management programs, such as 386-To-The-Max or QEMM386, generally make access to extended memory available for EMS emulators and the XMS (eXtended Memory Specification) interface. They also provide an option for loading TSR programs, device drivers, and parts of DOS above the 640K limit, which creates more space in conventional RAM. These programs and drivers are loaded, using the page table, into memory ranges between the 640K and the 1 Gigabyte limits, without being present there. The main trick is to determine which parts of this memory range are unused by hardware or other software, rather than to program the page tables.

Multitaskers

Recently PCs have reached the performance levels of mainframe workstations. Because of this, many DOS users are demanding more advanced features, especially multitasking. Actually, many users are more concerned with being able to switch between programs, than with multitasking.

Programs that make task switching possible are available for all types of PCs. One of the most popular programs is DesqView from Quarterdeck. However, the Task Swapper available in the DOS 5.0 Shell has been giving DesqView some competition. DesqView/386 and Windows 3 bring true multitasking to the DOS world. Both systems permit parallel execution of multiple DOS programs and can display output in separate screen windows.

These multitaskers, which are a type of operating system that's grafted onto DOS, can run other operating systems besides DOS. Since we're concerned about system programming, we won't discuss these systems in detail. Instead, we'll examine a system programming problem called "hardware virtualizing". The V86 mode has been extremely important to the solution to this problem. Virtualizing hardware presents the illusion of an independent PC to any program executed in a multitasking environment. DOS programs are inhospitable to other programs, writing directly to video RAM or programming the interrupt controller to reserve all available memory for themselves.

Hardware virtualization

The screen is one example of the basic problems that can occur in multitasking. Usually, each DOS program can assume that it can access the entire 80x25 text screen. However, multitasking provides a window smaller than the screen for each program. The multitasker places the output from each program into a buffer and displays a part of the buffer in the window assigned to the respective program.

To display all program output, the multitasker shifts the window contents. This displays other parts of the virtual screen. The virtual screen may also be expanded to use the entire physical screen, similar to Windows.

Intercepting video RAM access

Now that we know what happens from the user's perspective, let's see what goes on behind the scenes and how the multitasker handles the output from different programs. Problems shouldn't occur if the program sends output to the screen using ROM-BIOS video functions. The multitasker simply redirects the ROM-BIOS video interrupt to its own function. The multitasker's function stores the characters in the virtual screen (the internal buffer mentioned above), instead of writing the characters directly to the screen. The multitasker then passes the characters to the respective program window.

The functions in the multitasker handler intercept DOS video functions and high level language commands, such as PRINT, printf(), and WriteLn(), as well as ROM-BIOS functions.

Suppose the program doesn't support any of these options, and directly accesses video RAM (like most DOS based applications). Systems based on the 8086/8088 or 80286 processors can't be used because they don't provide an option for direct memory access. However, the V86 mode's paging mechanism allows you to control the memory access in 80386 and higher processors.

You may remember from the section on the 80386's paging mechanism the linear address space is modeled on the physical address space using the page table. This page table is vital to the multitasker. All DOS applications must be given the illusion they're operating in the first megabyte of memory, regardless of their physical locations in memory.

The paging mechanism also contributes to virtual screen management. The Present bit is part of this virtual management. This bit is part of the flag saved with each entry in one of the page tables. If the Present bit is unset during page access, the processor assumes the page currently doesn't exist in memory. An exception, which passes program control to the operating system, occurs. After program control is redirected, the page loads. The exception call occurs regardless of whether virtual memory is implemented.

A multitasker takes advantage of the exception by marking the pages, which correspond to video RAM, as not present. Each video RAM access calls the exception routine of the multitasker, which stores the video RAM access. The multitasker then places the byte (or word) to be displayed in the appropriate program's virtual screen for later display. This procedure saves us from having to redirect the video BIOS function. The video RAM directly receives the data, which inadvertently calls the multitasker's exception handler. Since this requires a lot of microprocessor time, a multitasker usually reroutes video BIOS for its own use. By using the procedure we just explained, we can protect both video RAM and other memory ranges from access. This also applies to individual memory locations and entire 4K pages.

Since the offset address of the access is always transmitted to the exception handler of the multitasker, it's even possible to protect ranges smaller than 4K. If the exception handler determines the memory location lies outside the range to be protected, it sets the Present bit to 1, executes the memory access, and changes the Present bit to 0.

I/O access interception

Hardware ports must also be handled if they're accessed simultaneously by DOS applications under multitasker control. Let's return to our screen output example. Think of how a video card changes to graphics mode or selects another color palette. The multitasker notices such changes through the I/O permission bitmap in Protected mode (see the "Protected Mode" section in this chapter for more information). This bitmap is always active in V86 mode and offers the multitasker the option of protecting ports from program access.

If a program tries to control one of these ports, an exception occurs, notifying the multitasker. Windows Enhanced mode uses this method when a DOS window in V86 mode calls the DMA controller during disk access. Paging doesn't help; the addresses are viewed as true physical addresses. So, Windows must intercept each access to this chip within the DOS window and convert the specified physical addresses into their real physical addresses before passing them to the DMA chip.

The V86 mode, the paging mechanism, and the I/O permission bitmap can solve many of the other problems facing multitaskers. However, the conflict that occurs in Protected mode between multitaskers and DOS extensions cannot be solved by any of them. If you start a program, which was developed using a DOS extension, from a multitasker, accessing the program in Protected mode isn't possible and accessing the program from V86 mode will run it on the lowest privilege level.

The DPMI and VCPI software intercepts provide a solution to this problem (see the "DPMI And VCPI" section in this chapter).

DOS Extensions

Memory must accommodate data and program code in extended memory, but this memory area must be kept separate. This is a typical situation in Real mode under DOS. Software drivers, such as EMS or XMS, grant access to more memory for data. However, this usually involves compatibility problems that require program reconfiguration. For years, software developers have been demanding options that enable DOS programs to use extended memory just like conventional RAM.

DOS extensions make these options available. These are development tools that are available from various software developers and are designed to work with specific compilers. Standard C compilers and other languages (e.g., Turbo Pascal) provide DOS extensions. In this section, we'll discuss DOS extensions and how they are used in the C language.

DOS extensions must be specially adapted to the compiler because they interact with the executable code generated by the compiler. DOS extensions attempt to permit DOS program execution in Protected mode on 80286, 80386, or 80486 processors. A program running in Protected mode can access all of RAM, instead of only the 640K specified by DOS.

How a DOS extension works

A short definition of how a DOS extension works is: Run a program in Protected mode and revert to Real mode before making DOS or BIOS calls. You can change code in an EXE file to run it in Protected mode, but this doesn't apply to DOS or ROM-BIOS. This presents some problems, as you'll soon see.

DOS extensions operate according to a fairly simple principle, but with a twist. DOS extensions include various tools that apply to the standard compiler (e.g., Microsoft C 5.1 or 6.0). First, a new starting address for the program is passed, followed by the program code of the DOS extension. The DOS extension built into the program copies the program into extended memory without executing it. Next, while still in Real mode, a global descriptor table appears, containing code and data segment descriptions of the program. The data needed for this is obtained from the OBJ file, which was generated by the compiler when the program was created.

The sequence in which the various segment descriptors are entered was established during program creation. The various segment references from the data and code segment were already selector entries that couldn't be changed. Adjusting segment references, such as with a DOS loader, isn't necessary when loading a Real mode program. The segment addresses are already defined in the segment descriptors and can no longer be changed.

The program is almost ready to run, although it still needs an interrupt descriptor table and the interrupt handlers used for calling DOS and BIOS functions. Once these are installed, the program can execute in Protected mode. The DOS extension switches the processor into the Protected mode and starts the program. While the program is active, the DOS extension runs in the background above various DOS and BIOS functions (more on this later).

If the program ends in the usual way (i.e., DOS function 4CH), the DOS extension regains control of program execution. It releases memory and reverts to Real mode, thus returning to DOS level. The user receives no indication that Protected mode is being used.

DOS extensions for the 80286, 80386, and 80486

DOS extensions can be divided into two areas: DOS extensions for the 80286 and DOS extensions for the 80386 and 80486. The second group uses the full capabilities of the 80386 processor and its successors, and achieves higher execution speeds than the 80286 could handle. To fully use the 80386/80486 extension, the source code of your program must be changed. However, an 80286 extension may not require source changes (more on this later).

The Demands of Protected mode

A DOS program in Protected mode first confronts changes to memory management, which is characterized by selectors, segment descriptors and descriptor tables. Segment addresses and constant 64K segments don't exist. The DOS extension solved most of this problem by accessing the compiler generated code before calling the program.

The DOS program encounters problems if it must execute DOS or BIOS calls, or trigger external device interrupts. None of these were designed to be executed in Protected mode. So, calling them would crash the system when loading the first segment address into one of the segment registers.

Many programmers insist their programs don't contain DOS or BIOS calls because they are designed to run under the XYZ system. However, even if a program doesn't directly call DOS or BIOS functions, many indirect calls still exist. If you use high level language statements and functions for screen display, file management, or reading the keyboard, these statements and functions are still DOS or BIOS calls at their lowest levels.

Now, let's see what happens if a software interrupt or the keyboard calls a BIOS or DOS function (e.g., a hardware interrupt) during execution. The DOS extension assumes control of program execution because it placed its interrupt handler in the interrupt descriptor table before the program started. The DOS extension provides its own interrupts to replace the DOS interrupts (20H, 21H, 24H, etc.), the BIOS interrupts (10H, 11H, 12H, 13H, 14H, 15H, 16H, etc.), and the hardware interrupts (08H, 09H, 0AH, etc.). The IRQ0 to IRQ7 interrupt sources usually trigger the same interrupts as the various Protected mode exceptions (interrupts 08H through 0FH). Most DOS extensions reprogram the interrupt controller to redirect hardware interrupts IRQ0 through IRQ7 to other interrupts. This keeps hardware interrupts and exceptions separate.

Let's return to the DOS extension's interrupt handlers as they are executed from Protected mode. These interrupt handlers are responsible for reverting to Real mode, calling the original interrupt handler from the interrupt vector table, then returning to Protected mode. The return to Protected mode indicates the end of the interrupt handler execution, after which the program can continue running. This momentary switch into Real mode remains completely transparent to the program itself.

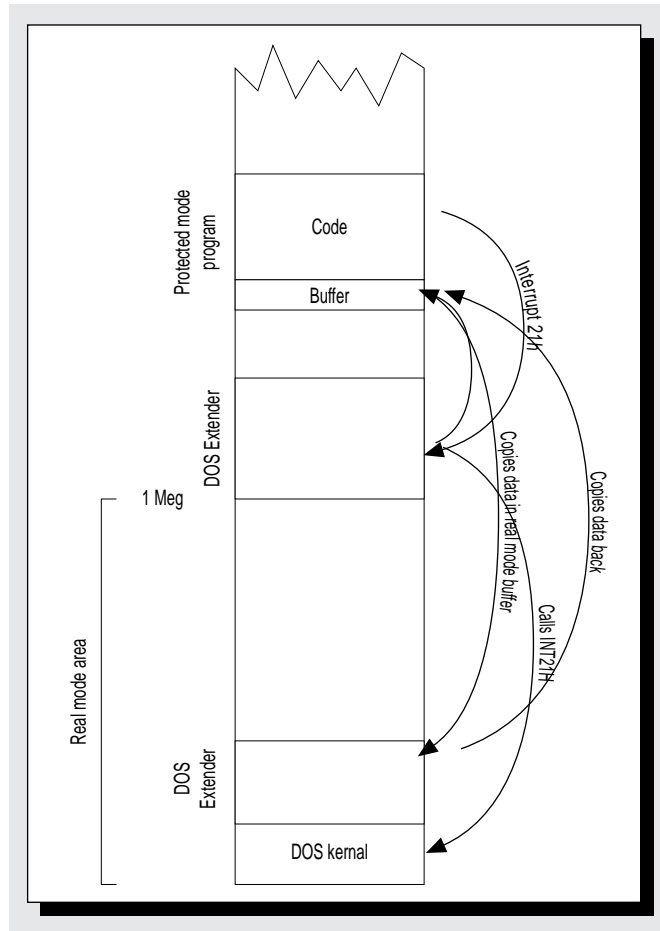
Switching between Protected mode and Real mode is time-consuming. This is especially true with the 80286, which directly supports only the switch from Real mode into Protected mode (returning to Real mode takes some effort). A modern AT with a clock speed of 16 MHz makes about 2000 loops from Real mode to Protected mode and back in a single second. An 80386 with a clock speed of 20 MHz makes nearly 9000 loops. This occurs because of the clock speed and the ease with which the 80386 returns to Real mode from Protected mode.

Both these numbers seem small when compared to the hundreds of thousands of assembly language instructions executed by a processor every second. However, fewer calls to DOS, BIOS, and hardware interrupts occur than you might think, especially in programs that access large quantities of data rather than interact with the user. These data access programs are good candidates for DOS extensions, because the standard 640K of RAM just isn't sufficient. So, the time loss between Real and Protected mode is minimal.

Transferring buffer addresses

Switching from Protected mode to Real mode doesn't affect many interrupt functions, especially interrupt 21H and its DOS functions. These functions expect parameters in the various processor registers and also frequently store data in these registers. The DOS extension's interrupt handler could pass this information to the original DOS interrupt handler unchanged, following the transition to Real mode, but a system crash would result. A crash will most likely occur when a call is made to a DOS function, which expects a buffer address as a parameter. This also applies to many DOS functions.

*Buffer transfer
on calling a
DOS or BIOS
function*



Before the DOS function call, the program loads the buffer address into the register provided. However, when the DOS function is called, the function finds a selector instead of the segment address it expected to find. Remember the selector and segment address don't correspond. The DOS function might access segment 104H, although what was actually intended was selector 104H (and any segment above the 1 megabyte limit).

The DOS extension converts the indicated Protected mode buffer references into segment references before calling the DOS function. However, a look at the associated global or local descriptor table reveals additional problems. The table provides a 24-bit or 32-bit base address, but no 16-bit segment address. The Protected mode address is rarely converted into a normal segment address, because the program and its buffers are beyond the 1 megabyte barrier, and beyond the reach of the Real mode functions of DOS.

Before the function call, the DOS extension must copy the transmitted buffer contents into its own temporary buffer, allocated below the 1 megabyte limit. This process must be repeated after the function call, in reverse order. If DOS changed the content of this buffer, these changes must be passed to the function caller. This means the contents of the temporary buffer must be copied back to the original Protected mode buffer.

Transferring the contents of buffers is time-consuming, especially when accessing files that contain large data blocks. However, this is actually where the recopying process barely matters because copying a buffer using a REP/STOSW instruction takes less time than accessing an external storage medium.

As described, the DOS extension masks the various DOS functions and BIOS interrupts that can be called by an application. TSRs and other system extensions that alter functions have done this for years in DOS. Copying data between Real mode and Protected mode buffers involves a lot of work and also takes time, but this isn't the biggest problem confronting the developers of DOS extensions.

Not all DOS functions expect a buffer address. The transition takes place in different buffers and the buffer length isn't consistently coded. The DOS extension must have an intimate knowledge of DOS functions, treating each function separately. The biggest problem occurs when an application uses undocumented DOS functions. The DOS extension may not recognize the registers, and may not be able to change to Real mode buffers. So, you should avoid calling undocumented DOS functions.

Similar problems arise with other, more frequently used software interfaces, such as the Microsoft Mouse Interface (interrupt 33H) and the NetBIOS functions (interrupt 5CH). If the DOS extension doesn't support these functions, the programmer must convert the selectors into segment addresses and the copying buffers. However, it's possible to use a few functions that are specifically provided by the DOS extension for this job.

Memory management

The DOS extension must act as a replacement for these functions, rather than simply an extension of the functions. This applies to memory management functions that usually control memory ranges below the 640K barrier. Almost all programs, even those produced by high level language compilers, use these functions to allocate memory for their own use. DOS never gives them more memory than is available up to the 640K barrier. If it did, most programs would use the memory immediately because it lies below the 1 megabyte limit and is addressable from Real mode. A program created with DOS extensions should be able to use all RAM. The program needs a way to identify the various memory ranges using selectors instead of segment addresses.

This is reason enough for a DOS extension to take over memory management instead of calling DOS memory management functions. These functions can then tell the program how much memory is still available, up to a complete megabyte. The number of free 16-byte paragraphs in BX limit that amount to one megabyte.

The next step is for the program to request the necessary memory, probably as far as the upper limit determined by the previous function call. The DOS extension may allocate one megabyte, then this memory is accessed by the appropriate selector. Because most programs make multiple memory requests for different purposes, this may make all physical memory available.

However, a problem occurs when you make memory ranges greater than 64K available. If the program code was written for an 80286 or one of its successors, it will work exclusively with 16-bit registers and, consequently, only 64K segments. When running through larger memory ranges, the program will occasionally be forced to increase the segment address to get past the first 64K of such a memory range. However, Protected mode prohibits segment address arithmetic, because new selectors, which point to entirely different memory ranges, are then created.

Making memory segments larger than 64K available in the 80286 creates problems for the DOS extension because a segment cannot accept more than 64K. Creating a larger memory block forces the DOS extension to access several segments and enter multiple segment descriptors in the global or local descriptor table. In turn, these descriptors require more selectors, which the DOS function doesn't return to its caller.

So that memory segments containing more than 64K can be allocated and accessed, in the case of the 80286, the interaction between a program and the DOS extension is read. This goes beyond the normal calling of a DOS function. We'll see how this actually works later in this section.

The absolute reference to specific segments also causes problems. Think of the interrupt vector table (segment 0000H), the BIOS variable range (segment 0040H), and the video RAM (segments A000H, B000H and B800H) that are addressed directly by many programs. The DOS extension can specify its own segment descriptors, with numbers identical to the corresponding segment addresses, although the video segments will cause some additional problems.

You must extend the global or local descriptor table, as needed, to $B800 + 1$ entries. This allows the video segment to be modeled by the descriptor with the number B800H. The descriptor table will exceed 360K, but most of the entries remain unused because no DOS extension needs B800H different segments for program execution.

DOS extensions form the cited segments using segment descriptors, which are appended to the original end of the descriptor table in question, and include an index with no relationship to the modeled memory segment. The DOS extension makes this selector available to its program functions. The memory segments can then be accessed easily.

DOS Extensions for the 80286

It's generally true that DOS extensions for 80386 and 80486 result in more efficient programs than DOS extensions for the 80286. However, the 80286 extension simplifies porting programs. In fact, porting with one of the best known of the 286 DOS extensions, DOS/16M from Rational Systems, is even restricted to recompilation under the Large memory model and subsequent attachment of the DOS extension. DOS/16M allows you to easily change a program for execution on a DOS machine that may have caused "insufficient memory" messages. After adding DOS/16M, the program can access more memory than before. This works as long as the developer follows the rules, especially in segment address interaction.

Creating a Protected mode program with DOS/16M

Program creation isn't very different for the compiler. The program is compiled in the Large memory mode to execute all data and code pointers as FAR pointers. So, data and code can exceed the 64K limit.

The OBJ file is initially still an ordinary Real mode program. The linking process combines the program, C libraries, and DOS extension object modules. These give the program a new starting routine and replace the various C compiler library routines with the extension's own routines. Memory is also allocated for the later attachment of the descriptor tables. The new starting routine ensures that program execution immediately terminates if the program is run on an 8086 system, or if insufficient extended memory exists. The resulting EXE file cannot run in Real Mode or protected mode. It must be sent through a conversion program named MAKEPM, which converts the EXE file into an EXP file. MAKEPM configures the segment addresses to the physical location of the segments in memory.

Remember that a Protected mode program requires selectors instead of segment addresses. MAKEPM replaces the various segment addresses with selector numbers. The sequence of these selector numbers indicates which segment descriptors will be entered in the descriptor table when the program starts. After MAKEPM processing, the file won't start from DOS. It must be run through a special loader. This loader loads the segments from the EXE file, places them in memory and stores their locations in the global descriptor table.

Two programs are needed: the loader from the DOS extension and the EXP program. Most DOS extensions include utilities that incorporate the loader into the EXP file. The DOS/16M utility is named SLICE, and creates an ordinary EXE file that can be started from DOS.

Altered library functions

Creating a Protected mode program is an interesting task because a Real mode compiler, such as Microsoft C or Turbo C, acts as the point of departure. There is only a slight difference between Real mode and Protected mode assembly language, if you overlook the selectors. Selectors are the same size as segment addresses, are stored in the same way, are processed by the same instructions, follow the same limits, and can be generated by a "postprocessor" like MAKEPM without accessing the actual program code.

However, the compiler library functions are changed dramatically when they're linked to the program code. Although a few functions remain intact, most are changed to avoid conflicts with the Protected mode rules. Among these are functions that change the program code, which is usually done under DOS for interrupt functions such as `int()` and `int86x()`. Other examples are the functions that handle segment arithmetic.

The `malloc()` function also changes. This is important to remember when porting C programs. `Malloc()` controls memory allocation in the C language, which is performed dynamically using the heap. Unlike the old `malloc()` function, the version included in DOS extensions can relinquish all extended memory to a program. In doing so, `mallo()` returns the selector and offset address of the allocated memory range. Since the `malloc()` function controls memory allocation, the DOS functions aren't used here.

Detecting pointer errors

Any program that extensively uses pointers will most likely encounter pointer errors. This is especially true for C programs; most crashes and errors in C code occur because of pointers that miss the intended memory object. If we assume that a single object was allocated using `malloc()`, such pointer errors can be easily detected from within Protected mode. The `malloc()` function specifies a new segment and a respective segment descriptor. This segment descriptor contains the size of the object as requested from the caller. If a pointer shoots past this segment, or accepts an invalid segment number, an exception occurs.

However, it's almost impossible to implement this procedure because the global descriptor table can accept only 8191 descriptors, many of which are also needed for other memory segments. Frequently using `malloc()` quickly fills the global descriptor table. It soon becomes impossible to allocate the remaining memory to the program. Although you could create a supplemental local descriptor table containing another 8192 segments, most DOS extensions rely exclusively on the global descriptor table.

`Malloc()` is forced to pack several memory ranges into a segment that increases with each call. Once the segment is full, or a requested memory block no longer fits, the next memory segment is established by appending a new descriptor to the global descriptor table. Different memory objects thereby share the same selector, even if their offset addresses are different. Problems occur only if a program wants to allocate more than 64K at the same time. The `malloc()` function in Real mode already allows this.

Despite the 64K limitation, it is fascinating how easily the data memory of an application can be expanded by changing the `malloc()` function. In the Real mode version, most C programs call `malloc()` until this function can allocate no more memory. Since Protected mode permits more memory, the process of allocating memory takes longer. This is the basis for the portability of normal C programs with an 80286 DOS extension.

It's still impossible to manage memory blocks larger than 64K, even though the DOS extensions offer special functions to handle this. For managing larger memory blocks, these functions simply create more sectors, one for each 64K segment. Obviously, it's impossible to run through such allocated memory ranges with normal pointer arithmetic. Instead, you must use special functions, which automatically switch to the previous or subsequent selector when a pointer approaches or overruns an offset part.

Additional options with the 80386

We've mentioned that programs created for the 80386 with a DOS extension execute more rapidly than those converted into a Protected mode program using an 80286 DOS extension. Now we'll learn why this happens.

DOS Extensions for the 80386

DOS extensions for the 80386 let you develop faster and more efficient Protected mode applications than you could with a DOS extension designed for the 80286 for many reasons, all of which involve the number 32. The 80386 is a 32-bit processor; this fact is important to high level language compilers:

- Segments can contain up to 4 Gigabytes. This makes it possible to eliminate juggling multiple code and data segments.
- 32-bit integers can be accommodated in a single register and processed by a single instruction.
- String instructions for copying, traversing, and processing memory ranges can operate on a DWORD basis, resulting in higher execution speed.

The following are other reasons why the 80386's abilities are better than the 80286's abilities:

- The paging mechanism lets the DOS extension provide the Protected mode program with virtual memory management that's transparent to the program. There are no memory restrictions.
- The addressing possibilities are expanded. Each register can now be used as an index register and multiplied automatically by a factor of 2, 4, or 8. This accelerates array access and gives the compiler the option of retaining more local variables permanently in registers.

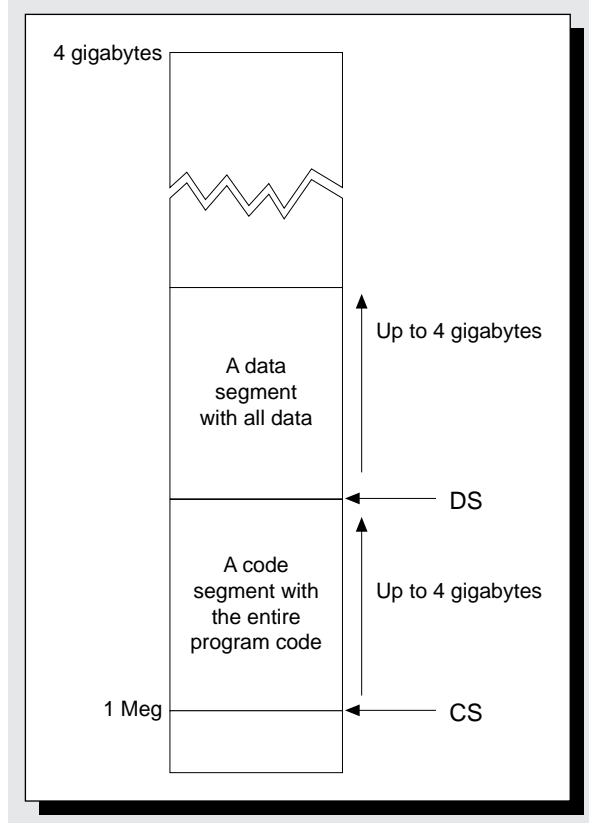
- The instruction set is expanded by the addition of efficient instructions for processing bit arrays. The compiler can use these instructions for processing bit fields and arrays for a dramatic increase in speed.
- Certain long distance jumps are also possible. Previously these could be made only by combining several instructions, which made the resulting program code larger and slower.

These options cannot be used with normal DOS compilers, which seldom support the 80286's expanded instruction set. Also, since pointer offsets are 32-bit instead of 16-bit, portability problems occur when using pointers.

The flat model

DOS compilers don't support the memory model used in most authentic 80386 compilers. This flat model is responsible for even greater increases in the execution speed of Protected mode programs. This model represents programs that consist of only two parts: a code segment housing all the program code and a data segment containing all constants and variables. The TINY model in DOS is similar, although the size of both segments is restricted to 4K. The 80386 flat model's segments combined can be up to 4 Gigabytes in length, which is more memory than most of us may ever need.

Flat model programs



Since the flat model uses NEAR pointers, faster execution is guaranteed. An offset address is needed, but a segment selector isn't necessary. The data offset address is found in the DS register, and the program code is in the CS register. The offset address occupies 32 bits. The flat model eliminates constant loading and reloading of the segment registers, as well as pointer arithmetic. So you can increment and decrement pointers like normal numerical words (dwords) without having to worry about changes in the selector.

Version 2.0 of OS/2 uses the flat model. Speed was one reason for this choice. The other involved portability. DOS extensions for the 80386 are frequently used for porting large programs from UNIX (programs used for mathematical or statistical

applications). In UNIX, however, memory segmentation is unknown and porting through a UNIX-compatible memory model is clearly simplified.

The compiler itself was often carried over from the UNIX side by using a DOS extension. UNIX compilers frequently have features that overshadow their DOS counterparts. However, using these features requires knowledge of the program, a lot of programming effort (resulting in large amounts of source code), and much compiler memory. The best known 386 compilers are the C386 compiler from MetaWare and the Watcom C/386 compiler by Watcom, Inc.

The best known DOS extension for the 80386 is the 386 DOS extension from Phar Lap. It works with both C386 and Watcom C/386. Its operation is barely different than the techniques described in conjunction with the DOS extensions for the 80286. DOS and BIOS calls in 80386 DOS extensions are frequently executed in V86 mode instead of in Real mode.

Calling DOS functions

Something has changed in DOS function emulation. The buffers, whose addresses must be given to many DOS functions, lie beyond the first 64K in the data segment of a 386 Protected mode program. So, these buffers cannot be reached by a 16-bit address, as DOS functions expect. Therefore, emulated DOS functions use 32-bit offset addresses and the expanded part of the register in question for the upper 16 bits.

Let's look at an example. DOS function 09H displays a string on the standard output device. To do this, it expects to find the address of the string to be entered in the DS:DX register pair. However, the emulated DOS function uses DS:EDX to process an offset address beyond the 64K limit. The DOS extension accepts the contents of the indicated buffer, but copies it into a temporary buffer below the 640K barrier. There the offset address is less than 64K, and the original DOS function is called, as usual, with the buffer address in DS:DX. The register expansion described here is used in both buffer address transfer and the handling of variable length data (e.g., in file functions).

The number of bytes to be processed are taken from the ECX rather than the CX register. Because the original DOS functions can only process 64K at one time, the DOS extension automatically divides its call into several 64K blocks. This repeats as needed, until the number of characters indicated was read or written. This is only one example of various functions that formerly processed 16-bit data, but can now accept 32-bit values.

Let's examine the Phar Lap DOS extension mentioned above. This extension can be used with another product from Phar Lap, 386[VMM]. This virtual memory management system makes almost unlimited (virtual) memory available to a Protected mode program. All memory barriers are gone, even though loading and reloading memory takes some time and thus affects performance. We recommend that you use this extension for programs that definitively require more memory than is available in most of the computers of the targeted user group. IBM Interleaf Publisher has also earned notoriety. This highly developed publishing system comes from UNIX. Before Phar Lap, it required a minimum of six megabytes of RAM; after Phar Lap it requires only two megabytes of RAM.

Using a virtual memory management system should be an integral part of program development. For example, anyone allocating an eight megabyte array, which is constantly being accessed from one end to the other, should realize that this array is going to slow program execution. DOS programmers must gradually learn how to work with 386 compilers because INT variables are now 32 bits wide instead of 16 bits. If you still want to work with 16-bit integers, you must use the SHORT type. Many functions (such as malloc()) were expanded.

DPMI and VCPI

The cliché "Too many cooks spoil the soup" also applies to Protected mode. Resource conflicts in Protected mode are similar to the cooperation or dissension between cooks. Protected mode utilities fight between two resources: extended memory and the processor's Protected mode. Each utility thinks it has all of extended memory available, and each utility believes that it can select the processor's operating mode.

A central authority doesn't exist to assume control of these operations. So, we find ourselves back in the world of DOS, where concepts like extended memory and Protected mode are unknown. For this reason, the different cooks cannot prevent one

another from acting. Nor can EMS emulators prevent the startup of programs developed with a DOS extension, or switch the processor into virtual 86 mode for multitasking.

The solution to this problem is important to system programmers, but not to the average user. From the user's viewpoint, he/she bought the 386 version of a program, and now must determine whether the program will work with the onboard memory management utility. The memory management utility is necessary because it keeps the RAM from filling with device drivers. However, the program refuses to work in multitasking operations.

Some progress has been made since the early days of Protected mode utilities. The participating software manufacturers defined software interfaces to ensure the peaceful coexistence of the various Protected mode utilities in the system. Two standards resulted: the somewhat older Virtual Control Programming Interface (VCPI), which is used in some DOS extensions and memory management programs, and the more recent DOS Protected Mode Interface (DPMI) used by Microsoft in Windows 3.

If we compare the two, DPMI easily wins. Although it's still too early to have found its niche in DOS extensions and other Protected mode utilities, DPMI seems more advanced than VCPI.

Discussing these interfaces in detail would exceed the scope of this book. So, in the following subsections we'll provide some general information about each interface and discuss the mechanisms used to keep DOS alive artificially.

VCPI

The VCPI interface was introduced in 1989 by a group of software firms under the leadership of Phar Lap (manufacturer of DOS extensions) and Quarterdeck (manufacturer of DESQView and QEMM). VCPI addresses the problems arising from the coexistence of DOS extensions, multitaskers, and memory management programs when these are run on an 80386 or 80486 machine. The 80286 plays no role in this specification.

The problem begins with installing a memory management program supplying the PC with virtual EMS memory. To use the paging properties of the processor, the DOS engine is switched into V86 mode and restricted to privilege level 3. If an application developed with a DOS extension is started on this plane from a multitasker, it won't be able to switch the processor into Protected mode or to install any sort of descriptor table. The processor's protective mechanisms perform this task.

These programs have the option with VCPI of switching into Protected mode and offer peaceful coexistence in extended memory. Most known memory management programs (QUEMM, 386-To-The-Max, etc.) support VCPI like the Phar Lap DOS extensions and the Quarterdeck DESQView multitasking environment. Newer versions of DOS, starting with DOS 5.0, also offer VCPI support which gives the EMM386.SYS device driver the job of providing access to the upper memory blocks.

Programmers developing an application with a DOS extension, or designing software for use under DESQView, rarely encounter the VCPI interface. The DOS extension or the DESQView API handles this. This interface is only important to programs that want to use Protected mode but don't have access to these utilities.

Client and server

If you work with networks, the terms client and server may already be familiar to you. These terms have somewhat different meanings under VCPI.

Server designates the application made available by the VCPI functions for use by the various clients. Under VCPI, the server is always a memory management program (e.g., EMM386.EXE), because these programs are installed by the CONFIG.SYS file on booting the system, and are present even before the first call to a DOS extension or multitasking environment.

The installed VCPI uses DOS extensions and multitaskers as clients located through the memory management system. A DOS extension or multitasker could work just as well as a server, though memory management programs are only active from the time the computer is started until it is reset. The other Protected mode utilities don't immediately assume this. Also, whatever utility gets there first has first priority, which favors the memory management programs.

VCPI services

The VCPI server provides a total of 13 different services to its clients, covering the ranges in which conflicts between the various Protected mode utilities can occur:

- Three functions dedicated to VCPI initialization.
- Four functions for rudimentary extended memory management.
- Three functions for access to CR0 (the first control register) and the processor's debug registers.
- Two functions for driving the interrupt controller.
- One function for toggling between V86 mode and Protected mode.

These functions were implemented as an extension of the EMS memory manager. Each memory management program must also make the EMS function interface available for EMS support. The point of departure is Version 4.0 of the LIM standard. The various functions can be called from EMS interrupt 67H, all functions carrying the function number DEH.

The VCPI is specified by a function code in the AL register ranging from 00H to 0CH. The client receives the result of the function call in the AH register, which is common in most EMS functions. A value of 0 indicates OK, and all other values indicate an error.

You can call interrupt 67H only from within Real mode or V86 mode. In Protected mode, the processor no longer uses the normal interrupt vector table, in which the pointer to the EMS handler is marked. Communication between the client and the server must be introduced in V86 mode by various function calls, which also give the client access to the server in Protected mode.

Initializing a client-server link

VCPI functions can only be used if a VCPI server is available. A potential client must first check for the presence of a VCPI server. This means the client must determine whether an EMS driver is available at all (see Chapter 12).

Next, an EMS page must be called using the normal EMS functions. This is done because many EMS emulators leave the computer in Real mode when no EMS memory is requested. It's only possible to recognize many EMS emulators as VCPI servers after switching to V86 mode. The EMS page must remain allocated until the end of the program to prevent the EMS emulator from returning to Real mode and switching off its VCPI services.

Function 00H checks for EMS emulator support of the VCPI interface. Placing DE00H in the AX register returns 00H in the AH register if proper support is available, and 84H in the AH register if only the normal EMS interface is supported. The BX register receives the VCPI version, with the version number in BH and the subversion number in BL.

Function 01H receives the VCPI interface. This is a relatively complex function, and can be understood only against the background of memory management in Protected mode.

Using the page tables

The VCPI server and the client running in Protected mode don't have descriptor tables. They have complete control over the GDTR, LDTR, IDTR, and TR registers. This doesn't cause conflicts because the descriptor tables contain linear addresses instead of physical addresses. They first come into play through the page tables by using the EMS emulator. EMS emulator operation is based on the use of the paging mechanism.

The VCPI client must use the paging mechanism, and may not model linear addresses directly on physical addresses by switching it off.

There's a good reason for this. Although the VCPI server and client may use the same (linear) addresses in their linear address segment, this is impossible in the physical address space. Doing so would mean using common physical memory ranges, and mutually overwriting each other in memory. Dividing the physical address space means either using a single page table

structure for server and client (and leaving the CR3 register constantly unchanged) or partially working with identical page tables.

This last option is handled by VCPI. The VCPI client must initiate its work with the interface by calling VCPI function 01H, which will fill its page table. This call must be made in Real mode, before switching into Protected mode. On calling the ES:DI register pair, the VCPI client must pass the physical address of a 4K range in memory, which it will use later as its first page table to the server, for modeling the first 4K of linear memory in physical memory.

The page directory, whose first entry to this page table must be denied, must be configured by the client before switching into Protected mode. The future page table is initialized by the server so the first 256 entries, which model the first megabyte of the linear address space on the physical address space, produce a 1:1 ratio between linear and physical address space. This means the client accesses the first megabyte of the physical address space using the first megabyte of its linear address space, the ROM-BIOS (which can access the video RAM), and conventional DOS memory.

This is how the VCPI server operates. It enables the use of this range for the exchange of data between client and server. However, the VCPI server can image further linear memory on physical memory with the aid of the indicated page table. VCPI function 01H returns the offset of the first entry in the page table in the DI register. From this address, the client can insert its own page table entries.

The length of the previously defined page table entries can be determined from the original offset address extracted from the value in the DI register. Because each entry takes up four bytes, the result of dividing the length by 4 is the number of pages already modeled, and the index of the next free page. Multiplying this index by 4K results in the first free address in the linear address space. This is important when allocating memory (more on this later).

Function 01H expects a pointer in the DS:SI register pair. Starting with this memory location, the server allocates three segment descriptors with a total size of 24 bytes, which the client must later add to its global descriptor table. It can copy the three descriptors for this purpose at any positions within its GDT, or reposition the GDT by the size of the memory range entered in DS:SI. It must also mark its position within the GDT and its index. This information will be used later as a selector to this first entry.

That describes the code segment by which the client is able to access the EMS functions of the server in Protected mode. However, here we're dealing with only one code segment instead of a CALL gate, so an offset address for the server call is needed. This is returned to the caller of function 01H, in the BX register.

Calling the server must be done using a FAR instruction referring to the selector of the server code segment. This indicates the value returned from the BX register as the offset address.

VCPI memory management

A VCPI client can usually assume the VCPI server has taken control of all available memory for itself and is to remain free. This memory is probably reserved for a RAM disk, a cache program, or something similar.

Calling BIOS interrupt 15H or the XMS interface is ordinarily useless to extended memory. Memory can only be requested by the VCPI client in Protected mode using the VCPI server, if you overlook the small fragment which the program is able to gain control of by using DOS functions, while still in Real (V86) mode.

VCPI memory management is based on the 4K pages the 80386 and its successors control by paging. Memory can be allocated and released in multiples of 4K. All four VCPI functions can be called in both V86 mode and Protected mode. This doesn't apply to all VCPI functions.

Function 02H determines the physical address of the highest page yielded by the VCPI server. Pass the value DE02H to the AX register. A value of 0H returned in the AH register indicates the call was successful. The required information will then be found in the EDX register.

Use this information carefully. Many VCPI servers produce the address of that last possible page instead of the address of the last physically available page. The result can be that function 02H will always return the value FFF000H (last page before the 16 megabyte limit), and not 3FF000H, if the computer has four megabytes of RAM.

You should avoid using function 02H. Instead, call function 03H. This function returns the number of pages available in the EDX register. This page number total can then be requested using function 04H, but only under certain circumstances.

With each call, function 04H returns the physical address of the required page if the value in the AH register is 00H. Any other value points to an error in page allocation and proceeds with an undefined value in the EDX register.

The VCPI client must do two things to access this memory:

- It must establish a segment descriptor in its global or local memory descriptor table to address the allocated memory using a segment. The segment can contain more than 4K if several pages must be combined.
- It must make an entry in the memory table by which the 4K from the linear address space can be modeled on their physical address. Naturally, the linear address recorded in the segment descriptor will be important. It determines the position of the associated entry in the page table. Because a page table was already established for the first four megabytes (it must be returned by function 01H), a linear address within the first four megabytes should be chosen. Don't forget the VCPI server, when calling function 01H, has already made some entries in the page table to image at least the first megabyte of the linear address space. The first free linear address and the index in the page table can be calculated, as we previously explained.

Function 05H frees pages if the VCPI client no longer needs them, but the client may no longer access them, even though this would seem to be possible through its page table. Besides the function number, the function expects the physical page address in the EDX register.

VCPI and EMS memory

Memory can also be requested by the VCPI client from the normal EMS functions by using a multiple of 16K. Access to these EMS pages occurs by focusing them in a page frame below the one megabyte barrier. VCPI function 06H returns the physical address of a 4K page below the one megabyte barrier. You can also discover memory ranges not normally occupied by RAM, which can be used by the memory management program through the page table. DOS Version 5.0 also uses this backfilling process for accessing the upper memory blocks.

The number of the requested page is placed in the CX register after the call to function 06H. This is returned by dividing the linear address by 4096, which is the same as rotating this address 12 bits to the right. Calling the function returns the physical address of the memory page at this address in the EDX register.

If the content of the EDX register is identical to the returned CX value multiplied by 4096, no alien memory is being used at the address in question. The linear address of this 4K page is different from its physical address. You cannot request the entire one megabyte address space in this manner, especially for those ranges occupied by the ROM-BIOS or BIOS extensions. In these cases, the function 06H returns error code 8BH in the AH register. Otherwise, a value of 00H in the AH register indicates no error.

Accessing debug and status registers

It's possible, at least in V86 mode, to access the various debug registers because the privilege level 3 is active. VCPI functions 08H and 09H read from and write to the various debug registers from V86 mode. These registers aren't very useful to a normal program, but are extremely useful to Borland's Turbo Debugger, which also uses the VCPI interface.

These two functions process all eight debug registers. The register contents are passed to a buffer to which the ES:DI register points when calling the function. Because each debug register occupies 32 bits, this buffer must have room for 32 bits. DR0 is stored in the first DWORD of the buffer, DR1 in the second, etc. Function 08H copies the debug register contents into the buffer, while function 09H loads them from the buffer.

Function 07H, which returns the contents of the CR0 status register in the EBX register, appears as an anachronism. Although the current operating mode and the paging mechanism status can be determined through this register, it's also returned by the nonprivileged SMSW instruction which loads the content of this register into another processor register or a memory location.

Intercepting and setting interrupt vectors

Earlier in this chapter, we recommended reprogramming the first interrupt controller for V86 mode exceptions. If you don't reprogram, hardware interrupts IRQ0 through IRQ7 encounter many exceptions.

VCPI function 0AH returns the base interrupt of the first interrupt controller in the BX register, and the base interrupt of the second interrupt controller in the CX register. If the two controllers no longer contain their default values (08H and 70H), the VCPI server has already rerouted these interrupts. Further rerouting is then strictly forbidden.

If the interrupt controller hasn't already been changed, it is possible to reprogram the first and second interrupt controllers. Of course, access to the two controllers must be done after all interrupts have been switched off.

VCPI function 0BH notifies the VCPI server about the reprogramming. This function must be called by the client before the interrupt flag is reset. Like function 0AH, the base address of the first interrupt controller must be entered in the BX register and the second in the CX register.

Changing the operating mode

One of the most important services of the VCPI server is the controlled transition from V86 mode into Protected mode and back to V86 mode. VCPI function 0CH performs this task.

This function can be called from V86 mode using the normal EMS interrupt. In Protected mode, function 0CH is accessible only through the code segment and the offset address returned by the VCPI server on calling VCPI function 01H. When calling this function from V86 mode, the linear address of a data structure, whose structure is shown in the following table, is expected in the ESI register. The following values are loaded into the various system registers:

Data structure for switching from V86 mode to Protected mode after calling VCPI function 0CH		
Function	Meaning	Type
00H	CR3 (Starting address of page directory)	dword
04H	Linear address of variables under dword one megabyte limit, contains the FAR pointer for the GDTR register	fword
08H	Linear address of variables under dword one megabyte limit, contains the FAR pointer for the IDTR register	
0CH	Selector for the LDTR register	word
0EH	Selector for the TR register	word
10H	Point of entry into the program code after switching into Protected mode with selector and 32 bit-offset address	fword

It's very important that this data structure be established beneath the one megabyte barrier. Only in this range are the address spaces of the VCPI server and its clients identical, as noted in connection with VCPI function 01H. The CR3 function is loaded with the value provided in the first step after calling this function. The page tables of the VCPI server are no longer valid; the page tables of the client are used instead. The server can now read data only from the coincident ranges, from the range below one megabyte.

Next, the GDTR register is loaded. The global descriptor table is referenced and may be anywhere in the address space of the VCPI client. This table must have been previously initialized in V86 mode and it's assumed the VCPI function was also previously called.

Now the LDT, IDTR, and TR registers are loaded and finally the switch to Protected mode is made. All interrupts are still blocked at this point and should remain so. Of the six segment registers, only the CS register is loaded with a valid selector

at this time. It's time to set up the program's own stack and load the DS, ES, FS, and GS registers with valid selectors. Only then may the interrupt lines again be opened. If an interrupt does happen to occur before the segment register is loaded, invalid selectors will be loaded. The processor will trigger an exception. Returning to V86 mode is effected by function 0CH, but this time with entirely different register loading. However, as the starting point, it's again necessary to obtain the code segment selector and the offset address returned by function 01H.

Before calling function 0CH, the VCPI client must establish a data structure in physical RAM below the one megabyte limit. The values which the VCPI server is to enter after switching into V86 mode are stored there. Using the entries for CS:EIP, the memory address is established for program execution in V86 mode. SS:ESP describe the new location of the stack.

Remember the values for the various segment registers do not concern selectors, but base addresses (physical address divided by 16), as is normal in Real mode and V86 mode. Should the program in V86 mode not have been developed especially for the 80386 or its successors, any values can be specified for the FS and GS registers because they aren't important to program execution. Although only one word is required for the various segment registers, two words are used in the data structure shown below. The respective segment address is entered in the first word, while the second word remains unused.

Data structure for switching from Protected mode to V86 mode after calling VCPI function 0CH					
Func.	Meaning	Type	Func.	Meaning	Type
00H	Reserved	dword	18H	SS after switching into V86 mode	dword
08H	EIP after switching into V86 mode	dword	1CH	ES after switching into V86 mode	dword
0CH	CS after switching into V86 mode	dword	20H	DS after switching into V86 mode	dword
10H	Reserved for EFLAGS register	dword	24H	FS after switching into V86 mode	dword
14H	ESP after switching into V86 mode	dword	28H	GS after switching into V86 mode	dword

For the VCPI server to access the above table after calling function 0CH, its linear address must be transferred to the SS:ESP register pair. It's also necessary for the client to load the DS register with a selector to a data segment descriptor which it has itself entered in its global or local descriptor table before calling the function. This descriptor must contain the base address 0 and have a size of one megabyte. After switching into V86 mode, the content of the normal registers (EBX, ECX, etc.) is retained. Only the content of the EAX register has been changed and is undefined. The Protected mode registers GDTR, LDTR, etc. have again been loaded by the VCPI server with the addresses or the selectors of its descriptor tables; but that doesn't in any way affect a program in V86 mode.

DPMI

The DOS Protected Mode Interface (DPMI) wasn't originally created as a general standard for collaboration between multitaskers and other Protected mode utilities. Instead, it initially appeared during the development of Windows 3.0 as an in-house product used for enabling the execution of Windows applications in extended memory. For unknown reasons, Microsoft decided to publish and expand this interface. A DPMI committee met in early 1990. The members of this group include Microsoft, Borland, Intel, Eclipse, IBM, Lotus, Phar Lap, Quarterdeck, and Rational Systems. Since many of these companies are VCPI supporters, we can assume the DOS extenders, multitaskers, and memory management programs of these manufacturers will also support both DPMI (DOS Protected Mode Interface) and VCPI in the future.

Differences from VCPI

Microsoft's marketing strength and the popularity of Windows are the principal driving forces behind this committee. The DPMI specification is superior to VCPI, in both concept and execution. DPMI is simply "cleaner" and more generic than VCPI. DPMI covers the entire spectrum of services required by DOS programs in Protected mode for peaceful coexistence with other Protected mode programs. These include the following:

- Managing descriptor tables of a protected mode program
- Management and allocation of extended memory
- Management of interrupts and exceptions

- Communication with Real mode programs and interrupt handlers
- Access to various processor registers
- DMA virtualization

These services show that DPMI clients must relinquish much of their Protected mode rights to the DPMI host. Unless the programs cooperate, this is impossible. The processor ensures that clients will use the services in this way. Unlike the VCPI server, the DPMI host runs on a higher privilege level than its client, giving it control of all clients and their activities. The lack of such protection was one of Microsoft's greatest complaints against the VCPI interface.

Clients and hosts

DPMI uses the term hosts instead of servers. The two words mean basically the same thing, and the change was probably intended to separate DPMI from the old client-server model. Despite this name change, DPMI has a host (program) which makes the DPMI services available to clients. However, DPMI has only one host, which is Microsoft Windows. Windows 3.X supports DPMI Specification Version 0.9 (the version the DPMI Committee released to the public). Version 1.0 was recently released, and has a few differences from Version 0.9 (more on this later).

16-bit or 32-bit DPMI

Unlike VCPI, DPMI was conceived for the 80286, even though Windows 3.0 doesn't support the 80286. Remember that DPMI becomes active in Windows enhanced mode, which is only available on 80386 or higher systems. As 80286 machines become less important, it isn't likely that an 80286 DPMI host will be released.

Because of the 80286 support, DPMI hosts exist in 16-bit and 32-bit versions. 16-bit hosts are designed for execution on 80286 computers, and support 64K memory segments. A 32-bit DPMI host works with 32-bit memory segments (up to 4 Gigabytes) and the resulting flat model. So, a host of this type runs only on a computer with 80386/80486 processors.

The target group

Like VCPI, DPMI is aimed primarily at the developers of DOS extenders and memory management programs. They should take advantage of the services of this interface to enable their Protected mode programs to run without difficulty even under a multitasking environment like Windows 3. The first DOS extenders that use them are already available on the market. Such a DOS extender, or the program generated by it, must also be able to function as DPMI host. If the program is started namely from the DOS prompt, without another DPMI host having been activated first, it must make these services available itself.

Execution on a virtual machine

The DPMI host always runs its clients on a virtual machine (VM), such as the DOS screen under Windows. A VM can run more than one DPMI client. For example, the memory management program can use the DPMI host services in a Windows DOS screen, and a program using a DOS extender can also be started from the DOS screen. As part of a virtual machine, the various programs in Real mode thereby divide up an address space, emulating a one megabyte DOS machine plus HMA. Multitasking between the various clients within a VM is possible between the various VMs. Then Windows can run several DOS programs in various DOS screens simultaneously. This becomes possible using preemptive multitasking, where the DPMI host interrupts program execution in a VM after a time to continue running a program on the next VM.

This occurs automatically when a time slice ends, although it's possible for clients to support this procedure by reporting they aren't busy, using the interrupt 2FH, function 1680H call. Once this happens, the DPMI host can pass execution on to the next VM, without the need to waste valuable processor time.

The biggest difference between Version 0.9 and Version 1.0 lies in management of the virtual machine. Under Version 0.9, all clients share a local descriptor table (LDT) and an interrupt descriptor table (IDT) within the VM. Clients can gain access to other memory segments (and perhaps other clients) by using the descriptors. The DPMI host cannot prevent this. However, this makes simultaneous execution of 16-bit and 32-bit clients impossible, because their respective segments cannot be merged into a single descriptor table.

DPMI Version 1.0 assigns each client its own LDT and IDT within a VM, which causes difficulties, especially in Windows. Many DOS Protected mode programs, which previously ran perfectly in Windows Protected mode from the DOS screen, are designed for the use of a common LDT with other programs, and require rewriting. Perhaps this why the Windows DPMI host continues to support only Version 0.9.

DPMI outputs and functions

The DPMI interface covers a series of output ranges. We'll provide a brief outline of these ranges, then list the most important data in more detail. We selected DPMI Version 0.9 as our standard. Only 13 different functions process the local descriptor table of the client and the segment descriptors contained there. This is quite different from VCPI, where the processing of these descriptor tables remains in the hands of the client, thus leaving the system open to many types of problems. Remember, though, that some DPMI functions still leave a few openings with potential for mayhem.

Each task under DPMI can access its own local descriptor table, kept separate from global descriptor tables. Global descriptor tables are reserved exclusively for the DPMI host. DPMI supports three different memory management function groups:

1. The first group allocates and accesses conventional memory below the one megabyte barrier. As we explained in the section on DOS extenders, Protected mode programs require this memory when calling DOS functions that expect to find data in buffers.
2. The second group handles extended memory. Memory blocks can be requested, freed, or altered.
3. The third group handles virtual memory. The 80286's lack of a paging mechanism makes this group unavailable on 80286 machines. This group includes functions for page maintenance, such as locking a page to prevent removal from memory.

A fourth group calls Real mode routines and interrupt handlers from Protected mode. By calling DOS or BIOS functions, the DPMI host saves its client a lot of work. However, you can also call a Protected mode routine from a Protected mode interrupt handler using callbacks.

The DPMI specification also includes functions supporting the debug register in the 80386 and its successors, for requesting and setting Real mode or Protected mode interrupt handlers, for blocking hardware interrupt releases, and for initializing a client and switching into Protected mode.

Interrupt 2FH call in Real mode	
1680H	Client unoccupied, pass on program execution
1686H	Query operating mode (also Protected mode)
1687H	Query DPMI availability status

The following tables summarize the various services offered to the DPMI client by the DPMI host.

Interrupt 31H call in Protected mode			
LDT memory management			
0000H	Allocate LDT segment descriptor	0007H	Establish segment base address
0001H	Release LDT segment descriptor	0008H	Establish segment length
0002H	Image Real mode segment on segment descriptor	0009H	Establish access rights/segment type
0003H	Query increment for selector	000AH	Create alias for a code segment
0004H	Block removal of segment	000BH	Query segment descriptor
0005H	Permit segment removal	000CH	Load segment descriptor
0006H	Query base address of segment	000DH	Request specific selector
DOS memory access			
0100H	Request DOS memory	0102H	Change size of a memory block
0101H	Release DOS memory		

Interrupt and exception management			
0200H	Return address of a real mode interrupt handler	0205H	Install protected mode interrupt handler
0201H	Set real mode interrupt handler	0900H	Block virtual interrupt flag
0202H	Query address of an exception handler	0901H	Release virtual interrupt flag
0203H	Install exception handler	0902H	Query virtual interrupt flag
0204H	Query address of a protected mode interrupt handler		
Calling real mode routines			
0300H	Simulate Real mode interrupt	0303H	Create callback
0301H	Call Real mode routine	0304H	Return callback
0302H	Call Real mode routine		
Miscellaneous functions			
0400H	Query version number		
Functions for accessing extended memory			
0500H	Request memory use data	0503H	Change size of memory block
0501H	Allocate memory in extended memory	0800H	Convert physical address into linear address
0502H	Release memory block		
Functions for managing interrupts and exceptions			
0600H	Protect memory range from removal		
Unlock memory region			
0602H	Unlock Real mode memory region	0702H	Privilege memory region during removal
0603H	Relock Real mode memory region	0703H	Mark memory range as overwriteable
0604H	Query page size		
Support for the debug registers of the 80386 and successors			
0B00H	Define breakpoint	0B02H	Get breakpoint status
0B01H	Delete breakpoint	0B03H	Reset breakpoint status

Client initialization and switching into Protected mode

DPMI calls begin with querying the DPMI host. The services cannot be accessed until this query occurs. The query must occur in Real mode (after the potential client starts at the DOS level).

So you must use function 1678H, which the DPMI host clicks into the DOS multiplexer interrupt 2FH during initialization. If it returns a 0 value in the AX register following its call, a DPMI host is installed. Status data on the DPMI host is then found in the other processor registers. For example, bit 0 of the BX register indicates whether the DPMI host is 16-bit or 32-bit. If bit 0 = 0, the DPMI host is 32-bit. However, the CL register indicates the processor type and the DX register indicates the version number of the DPMI host. The information in the SI register is also very important. SI conveys the memory block size in paragraphs required by the DPMI host for management tasks. The client must allocate this block size before the switch to Protected mode can occur.

The switch itself occurs through a routine whose address is returned in the ES:DI register pair once the multiplexer function is called. A jump must be made to this address using a FARCALL assembly language instruction. The segment address of the memory block made available to the DPMI host is placed in the ES register. If the DPMI routine called from ES:DI returns with the carry flag set, the program remains in Real mode because the switch to Protected mode failed for a reason not described in detail. But if the carry flag is unset, the program moves into Protected mode and can then call all DPMI services from interrupt 31H. Remember that most DPMI functions trigger an error during execution by setting the carry flag, but return no special error code beyond that. This, too, is a point of departure for future DPMI expansions.

Even after switching into protected mode, the program is still obviously below the 640K barrier. However it runs in an entirely normal manner, because the existing segment addresses in CS, DE, and SS were replaced by selectors. They point to memory segments of 64K each, whose segment descriptors were automatically established by the DPMI host. The selectors in DS and SS are identical, if the two segments were already coinciding in Real mode.

Besides that, ES points to a segment descriptor describing the PSP of the program and records 100H bytes as the segment length. GS and FS, if present, are also loaded with the value 0. After the successful switch into Protected mode, a DOS extender, for example, will attempt to load the program it created into extended memory from the hard drive, to run it there. However, to do this, it must first allocate extended memory from the DPMI host and enter (or cause the entry of) the associated segment descriptors in its local descriptor table (more on this later).

Calling DOS interrupt 21H, function 4CH from within Protected mode ends execution of the Protected mode program. The DPMI host then switches back to Real mode automatically, removes the original Real mode program (the loader) from memory, and returns to the normal DOS prompt.

Managing the client's local descriptor table

Since it has 14 functions, the local descriptor table can be difficult to manage. You'll rarely use these functions because the memory management area of the DPMI simply divides extended memory into memory blocks, but doesn't specify segment descriptors for them. The client is responsible for this. Also, the client cannot access its descriptor table directly for reasons of privilege. Instead, the client must use the DPMI host functions.

Function 0000H lets the client request one or more segment descriptors in its local descriptor table. In this context, the word "request" means the host establishes a desired number of data segment descriptors in the local descriptor table, each descriptor being first supplied with the starting address 0 and a corresponding length. Only additional DPMI calls allow the client to specify the desired starting address, segment length, and type, in case the client requires code segment descriptors instead of data. As the result, function 0000H returns a selector to the created segment descriptor. If several segment descriptors were requested, their numbers can be determined by addition of the value returned by function 0003H. Because of the complicated structure of selectors, you shouldn't assume, however, that it's always necessary to increment the returned selector by the value 1 to determine the following selectors one by one.

Before ending the program, all the selectors requested in this manner must be returned. Function 0001H performs this task for one sector. If you want to address a memory segment beneath the one megabyte barrier while in protected mode, you'll need a segment descriptor and an associated selector. Passing the Real mode segment address to function 0002H produces a segment descriptor for a 64K data segment. This segment returns to this descriptor. A segment descriptor taken over by the client using function 0000H or 0002H can be modified with the help of functions 0007H, 0008H, and 0009H. These let you specify the starting address, the segment length, access rights, etc. Regarding the segment length and starting address, the client must ensure that memory segments that aren't allocated yet or allocated memory segments that are covered by several, overlapping segment descriptors aren't accessed. The functions named simply aren't responsible for tests of this sort. This is the back door we mentioned earlier.

These data can be read using the LSL (Load Segment Limit) and LAR (Load Access Rights) instructions, or function 0006H can return the base address of a segment. Function 000BH can read a segment descriptor. Function 000CH loads a complete segment descriptor into a buffer. The DPMI ensures the client isn't granted a priority higher than 3. Function 000AH generates a corresponding data segment descriptor for a specific code segment. This descriptor contains the starting address and code segment length, returning a selector.

Extended memory allocation

Functions 0500H, 0501H, 0502H, and 0503H manage and allocate extended memory. Function 0500H returns status data. This data includes free memory available and swap file size on a virtual memory management system. Function 0501H allocates extended memory. After the function call, the BX:CX register pair contains the extent of the requested memory range available. Because the DPMI system is divided internally into 4K blocks due to virtual memory management (nearly always implemented in the case of a 32-bit host), we recommend that you request blocks in multiples of 4K. If the call was successful,

the BX:CX register pair receives the linear address of the allocated memory block. The SI:DI register pair receives a handle required for further block processing by the various DPMI functions.

Once the client takes possession of the memory block by calling function 501H, the block is still inaccessible. First, the client lacks the associated segment descriptor mentioned earlier. Function 0000H must be called to allocate the block, then fill the block with the transmitted starting address and the known length. The client now has the option of subdividing the required memory range into several contiguous segments. This is useful because DPMI doesn't have a type of "garbage collection" working in the background to combine memory blocks automatically that have become free to form larger units. A DPMI client should allocate all the memory required using function 501H.

Function 0502H returns a memory block from extended memory. The handle returned by this function identifies the memory block for the DPMI host (very important). Function 0503H changes a previously allocated memory block. The block's size (not its contents) is changed by this function, which passes the memory block's handle and its new size. If the call was successful, the new starting address is returned along with a new handle (if needed). If the memory block was in fact moved, the client is then responsible for bringing the start into agreement with the segment or segments created for gaining access to the memory block.

DOS memory allocation

Functions 0100H, 0101H, and 0102H allocate and manage DOS memory. These functions are similar to standard DOS functions 48H, 49H, and 4AH. Unlike the extended memory functions, these functions create a segment for access to the memory block allocated immediately, making it unnecessary for the client to perform the same task. Function 0100H requests the DOS memory. It requires the size, in paragraphs, of the desired range. If the call was successful, the function returns the selector to the allocated block's segment descriptor selector. Function 0101H releases the DOS block requested by function 0100H. Function 0102H lets you change the size of a memory block. Pass the selector in the DX register, and the new memory block size (in paragraphs) in the BX register.

DOS memory blocks are needed by DOS extenders, in particular, when DOS function calls issue from Protected mode, which results in the transfer of data into buffers. This is frequently the case when accessing files.

Virtual memory management

Functions 0600H, 0601H, 0602H, 0603H, and 0604H manage virtual memory. These functions are implemented only with 32-bit hosts (the 80286 doesn't have page tables or virtual memory management). These functions lock and unlock memory ranges. Function 0600H locks the specified memory range from removal. This prohibits the removal of interrupt handlers. Function 0601H unlocks locked memory ranges.

Functions 0602H and 0603H target RAM below the one megabyte barrier, which was allocated using function 0100H. Function 0602H unlocks pages for removal, and function 0603H locks pages. As the last function in this group, function 0604H returns the size of a page to the caller.

Interrupt handling and calling Real mode routines

Interrupt handling, particularly the reaction to hardware interrupts, represents one of the greatest difficulties in the Real mode programming of DOS extenders and other Protected mode utilities. How should a program react if an interrupt, normally available in Real mode, occurs in Protected mode? VCPI leaves this question to the programmer, but DPMI offers various solutions. Before we discuss these functions, however, let's briefly look at the concepts behind them.

First, no problems exist in DPMI between the various exceptions and the hardware interrupts IRQ0 through IRQ7. The host reprograms the first interrupt controller in each case. The lower privilege level of the client keeps the client from doing the same thing. The DPMI host also ensures that all hardware interrupts land in Protected mode, even if they occurred during program execution in Real mode or V86 mode.

The DPMI standard handler initially receives control over program execution, passing control to the first client that has recorded a handler for this interrupt through an appropriate DPMI function. If it returns to the caller through an IRET assembly language instruction, this is completed during interrupt handling. The interrupt handlers of the other clients are no longer used.

So, the existing handler should be queried with an appropriate DPMI function before installing a hardware interrupt handler, and called within its own interrupt routine.

When all interrupt handlers operate according to this scheme, the result is a sort of chain, with every interrupt handler getting its due time. Each DPMI client can protect itself against the release of hardware interrupts with certain functions. The DPMI host maintains a virtual interrupt flag for each client. This flag, which is stored in the Flag or EFlag register, ensures that no hardware interrupts can reach one client, while allowing passage to another client.

The situation changes with software interrupts triggered in Real mode. Only three of these interrupts reach protected mode, if the client makes corresponding handlers available: the BIOS timer interrupt (1CH), the **CmC** interrupt (23H), and the critical error interrupt (24H). However, software interrupts can also be triggered in Protected mode when, for example, a DOS extender enters an INT instruction in assembly language. If no DPMI client specified an interrupt handler for the respective interrupt, such a software interrupt results in a switch to Real mode, execution of the interrupt, and a subsequent switch back to Protected mode.

There's a problem with this switching. These functions generally receive data from the processor registers. There's no problem in the general registers, but the contents of the segment registers are destroyed during the shift into Real mode because they contain selectors instead of segment addresses. Because of this, DOS extenders break into the various software interrupts and convert the selectors in the segment registers into segment addresses while still in Protected mode, before passing the interrupt on to Real mode.

DPMI interrupt handling

Six different functions manage Real mode interrupt handlers, Protected mode interrupt handlers, and exception handlers. These functions, number 0200H through 0205H, install handlers, query handlers, and release handlers. Also, the DPMI host uses function 0300H to simulate a Real mode interrupt from within Protected mode. The Protected mode software interrupt handlers intercept the various software interrupts (BIOS, DOS, etc.), check the indicated function number, and, on the basis of this data, convert the pointers returned in processor registers into Real mode format after previously copying the data addressed into a DOS buffer below the one megabyte barrier.

Functions 0301H and 0302H call Real mode routines residing below the one megabyte limit. Function 0301H ends with the VAR RET assembly language instruction, while function 0302H ends with the IRET instruction. You can also simulate a Protected mode routine from within Real mode. For example, the mouse driver can allow a routine call when the mouse moves or button status changes. The routine must be performed in Real mode, but in certain cases, the mouse event should be readable from the context of a Protected mode program.

The DPMI host uses function 0303H for creating a callback. This callback is a short routine created independently in RAM by the DPMI host. When the callback is called, the desired Protected mode routine executes, if the switch to Protected mode has occurred. Functions 0900H, 0901H, and 0902H manage the virtual interrupt flags, protecting your program from unauthorized hardware interrupts.

Access to debug registers

To permit each DPMI client to access the debug registers of the 80386 and its successors, the DPMI host centralizes register access in four different functions. Each client can install its own breakpoints (sometimes called watchpoints) whose addresses are loaded by the DPMI host into the processor's debug registers, as soon as the client reaches execution.

Function 0B00H defines a breakpoint. The function requires the parameters needed to configure the debug register: the linear address of the breakpoint, its size (bytes, word, or DWORD), and its type (read, write, or execute memory location). It then returns a breakpoint handle, if too many break points weren't installed or an invalid parameter wasn't indicated on calling the function. Function 0B01H deletes a previously defined breakpoint. The function requires the handle of the defined breakpoint (function 0B00H returned this handle). Function 0B02H queries the status of a breakpoint. This indicates whether the breakpoint was triggered. Function 0B03H resets breakpoint status, which permits execution of the next breakpoint.

Then you learn whether the break point was already triggered. The corresponding flag can then be reset with function 0B03H so a new execution of the breakpoint can be detected.



CD-ROM And Its Technology

Today's most popular storage medium is the CD-ROM. Already several major computer manufacturers deliver only CD-ROM equipped PCs. CD-ROM titles which are available today number in the thousands, whereas these numbered in the hundreds only two years ago.

Developers have been adapting many hard drive based software applications to run on the CD-ROM drives. Some applications take advantage of the easy distribution method offered by CD-ROMs while other applications take advantage of the huge amount of storage space and special properties the CD-ROM offers over conventional drives.

In this chapter we'll talk about the structure of a CD-ROM and the different CD formats. We'll take you from the smallest bit all the way to the directory structure. You'll learn how to use software to link CD-ROM drives to a DOS system and how to control the drives directly using the appropriate drivers.

CD Formats

To understand the technology of CD-ROMs there's a few basic questions that we can first answer. For example, how are the individual bit sequences stored on the CD and how are they combined into logical blocks? What about the file system which groups the blocks into files and records information about the stored files in directories? We'll answer these questions in this chapter.

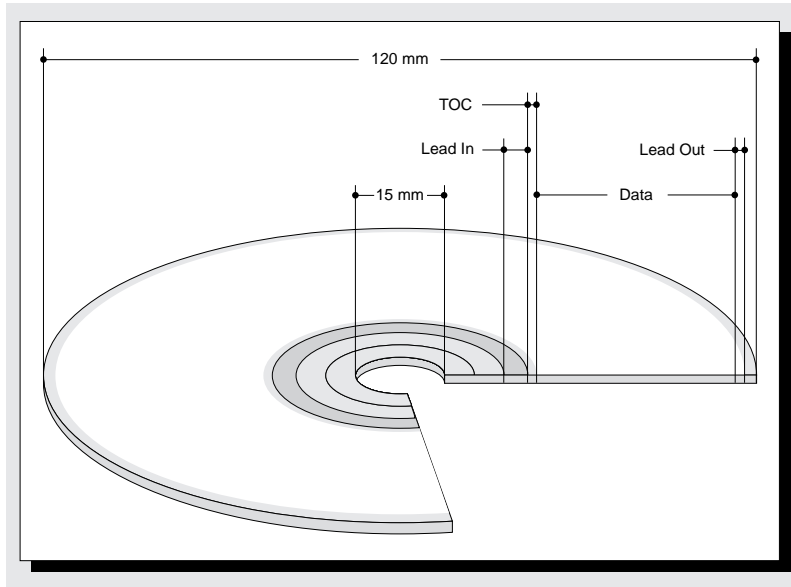
When talking about CD formats our starting point is the Red Book. The Red Book was published by Sony and Philips, the inventors of the CD, in 1982. This book is a detailed technical specification that describes the format of the audio CD in all its parts.

Other CD formats are based on the audio CD format and are specified by other books. For obvious reasons, these are known as the "Rainbow Books". The Yellow Book is concerned with the CD-ROM format, the Green Book contains the format for CD-I and the Orange Book describes the format for recordable CDs and the Photo-CD. However, all these formats are based on the Red Book.

The physical format

We'll start with the physical format which is identical for all types of CDs. Regardless of whether a CD is to be used for audio or data, the platter is 4.75 inches in diameter and 1.2 millimeters thick. The hole in the center of the CD has a diameter of 15 millimeters. The CD has a reflective layer of aluminum which is coated by a protective layer of clear paint.

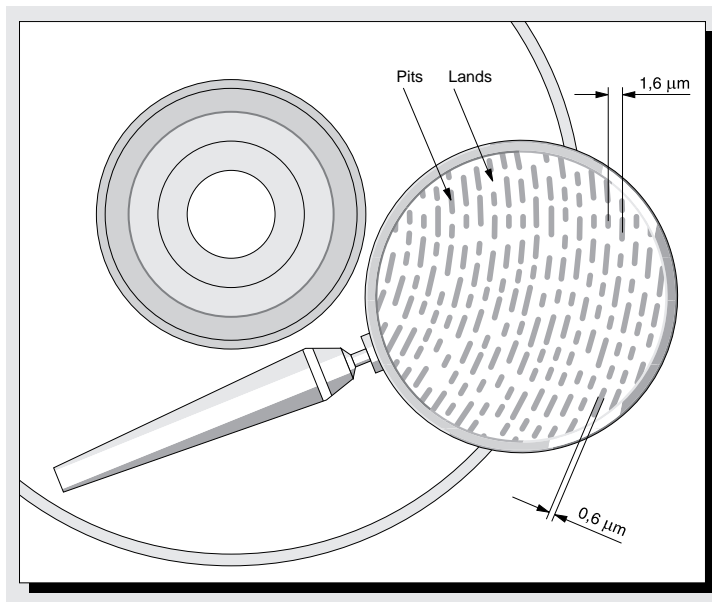
*Dimensions and
structure of a
CD*



When the CD is manufactured, the information to be stored on the CD is pressed into the layer of aluminum in the form of pits (indentations) and lands (elevations) which represent the individual bits. The pits and lands are arranged along a single spiral which covers the entire CD, winding from the inside to the outside. Unlike records, CDs start at the inner edge instead of the outer edge.

Because the pits are only 0.6 micron wide (a micron is the equivalent of one millionth of a meter), the path this spiral are separated by a microscopically small distance of only 1.6 micron. The track density is almost 16,000 tracks per inch (TPI). Compare this to the 135 TPI on a 3.5-inch HD diskette.

*Pits and lands
located on a
CD-ROM*



If this spiral were stretched out in a straight line it would be approximately 3.75 miles (6 kilometers) long. It also includes no fewer than 2 billion pits. Naturally, the laser beam that reads these pits and lands must be correspondingly small. The scanning beam is approximately one micron in diameter which makes it only a little larger than the wavelength of the light that forms its beam.

How a CD-ROM is organized

The recordable surface of a CD-ROM is divided into the following three sections:

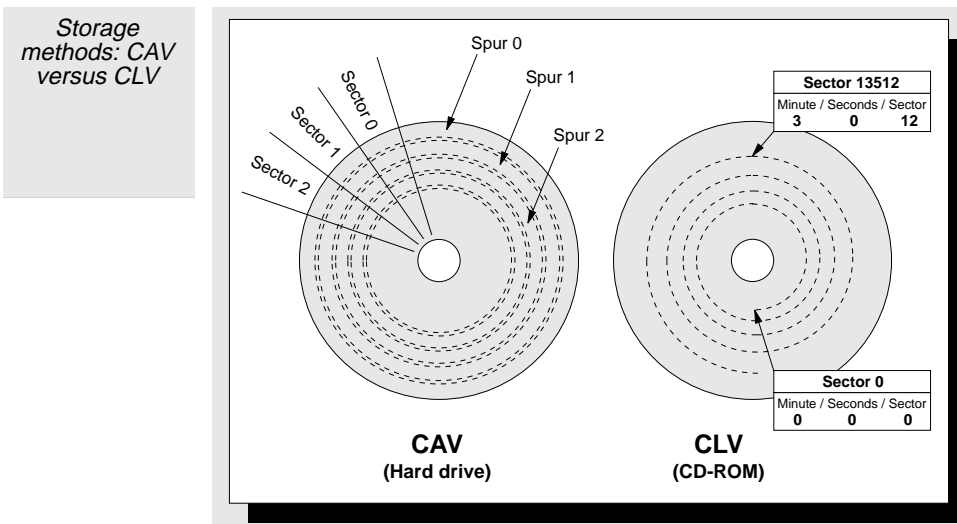
- The lead-in
- The data area
- The lead-out

The lead-in occupies the first four millimeters of the CD's inner edge and contains a type of table of contents. The lead-in is followed by the data area, which can occupy up to 33 millimeters, depending on how much data is on the CD. Finally, the lead-out range marks the end of the data. It follows immediately after the data area and is approximately 1 millimeter wide.

CAV and CLV methods of storing data

The two methods of storing data on rotating mass storage systems are called CAV (Constant Angular Velocity) and CLV (Constant Linear Velocity). Both names refer to the rotation speed of the storage medium.

Hard drives and diskettes that are divided into individual tracks and sectors follow the CAV principle. This is based on constant angular velocity. Regardless of where the read/write head is located above the medium, the medium always rotates below the read/write head at a constant speed. If the read/write head is above a track at the inner edge of the medium, it travels a much shorter course than it would over an outer track. Today's hard drives take advantage of this factor by packing more sectors into the larger areas of the outer tracks.



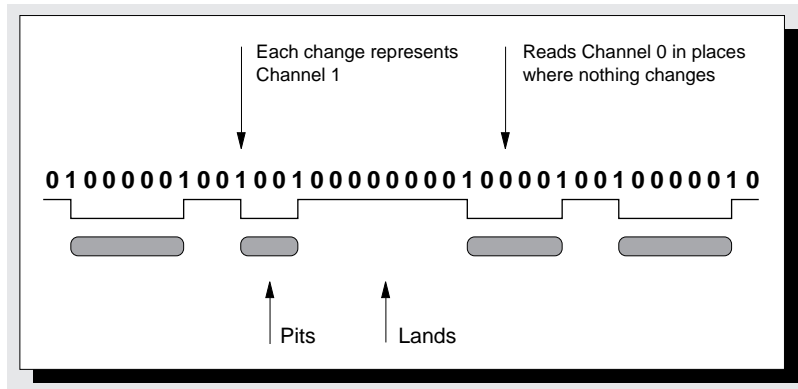
Perhaps the most important difference between the CAV method and the CLV procedure is the rotation speed. The rotation speed of the medium doesn't change with CAV. This is true regardless of the read/write head location. The opposite is true with the CLV method used by CDs. The read/write head for CLV always travels a constant distance in a specific unit of time regardless of whether the head is at the inner or outer edge of the CD. However, the rotation speed must be changed based on the position of the head (rotation speed = angular velocity).

Therefore, the rotation speed of the drive increases as the head moves from the inner edge of the medium to its outer edge. This is one of the reasons why a CD-ROM drive has significantly slower access times than a hard drive. It must constantly change its rotation speed. The time to speed up and to slow down becomes significant. Also, it's much more difficult to find a sector along a 3.75 mile long spiral than it is to find the same sector on a medium which is neatly organized into tracks and sectors.

Storage of bits and bytes

Since all the data on a disc is represented in a single continuous spiral, you might think of it as a data channel. On a CD, the bits are referred to as 'channel bits'. The transition from a land to a pit or the transition from a pit to a land is used to represent binary 1. Land and pits are used to represent binary 0. The length of a land or the length of a pit determines how many binary 0's are represented. This is the same procedure used to record data on magnetic storage devices like hard drives. The only difference is that magnetic flux change replaces the pits and lands.

A sequence of zeros and ones using pits and lands



Due to technical limitations, the minimum length of either a land or a pit is 3 bits; the maximum length is 11 bits. From this you can see there's a problem trying to represent two consecutive 1 bits when the technical limitations require no less than two and no more than 10 binary 0's between transitions.

Using this scheme, it isn't possible to represent all combinations of 0 and 1. Instead a scheme called EFM is used. EFM stands for Eight-to-Fourteen-Modulation. Using EFM, a byte that is to be stored is converted from its normal eight bits into 14 channel bits. The sequence of channel 0 and channel 1 within these bits is set by a simple conversion table which is part of the control unit of every CD-ROM drive. The codes within the EFM table are chosen so they avoid two adjacent channel 1 bits and stay within the maximum length of 11 channel 0 bits.

Par of the EFM table (EFM = Eight-To-Fourteen-Modulation)

Byte		EFM-Code
Dec	Binary	
0	00000000	01001000100000
1	00000001	10000100000000
2	00000010	10010001000000
3	00000011	10001000100000
4	00000100	01000100000000
5	00000101	00000100010000
6	00000110	00010000100000
7	00000111	00100100000000
8	00001000	01001001000000
9	00001001	10000001000000
10	00001010	10010001000000

However, even conversion by EFM table doesn't consider the separation of individual bytes. When the first byte ends in a channel 1 and therefore changes from pit to land (or vice versa), the next byte cannot again start with such a change, because there is no room between the two. Therefore, each byte with its 14 channel bits is extended by three additional channel bits called *merge bits*. Merge bits separate the bytes from each other and increase the number of channel bits to 17 per byte.

Frames

Now let move onto the smallest contiguous information block of a CD. This is called the *frame*. A frame includes 24 bytes (each having 17 channel

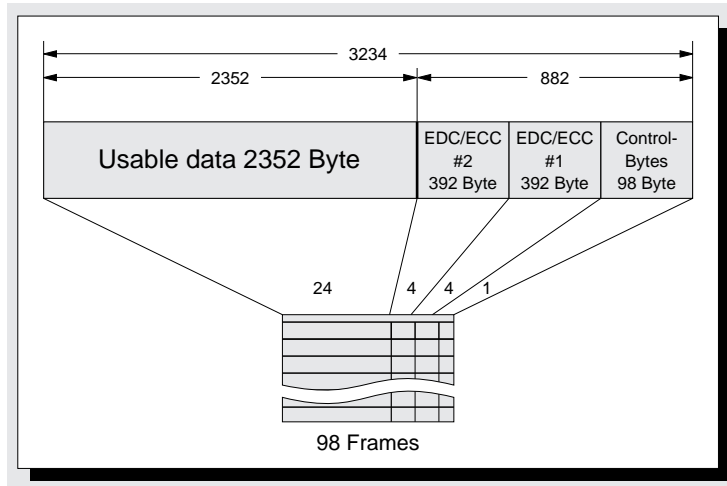
bits). Other information is added, which establishes the frame as a data block. This block begins with a sync pattern, a specific pattern of 27 channel bits which indicate the beginning of a new frame to the drive. This is followed by a control byte and then the 24 bytes of actual data. Finally, an additional 8 bytes for error detection and error correction, (again each having 17 channel bits) are appended to the frame. From the table on the right, you can see that there are 588 channel bits per frame:

Channel bits	Sync pattern
27	Control byte
1 * 17	Data
24 * 17	Error correction & error detection
8 * 17	588 Channel bits per frame

Sectors

At the next level, 98 frames comprise one sector. The following illustration shows that a sector consists of 3,234 bytes. A total of 2,352 of these bytes are available as usable data and the remaining 882 bytes consist of the data for error correction and detection and 98 control bytes. All audio CDs are based on this original CD-DA format.

A CD-DA format sector



This format is for the CD-DA format (CD Digital Audio) for audio tracks. Knowing that a CD audio player reads 75 of these sectors per second, we can draw some important conclusion about the operation and data transfer rates of a CD system. A CD player works with a scanning rate/frequency of 44.1 KHz, 16-bit samples and 2 channels, you can see that this throughput is 1,411,200 bits per second ($44100 \times 16 \times 2$). This equals 18,816 bits per 1/75 second, which again equals the 2,352 bytes of usable data that a sector can store according to CD-DA format.

Sectors are played back in a specific time. Addressing the data on a CD is based on time units in the format: minutes/seconds/sector. For example, the time unit for the fifteenth sector of the third second in the 13th playing minute of the CD is 13/03/15.

Subchannels

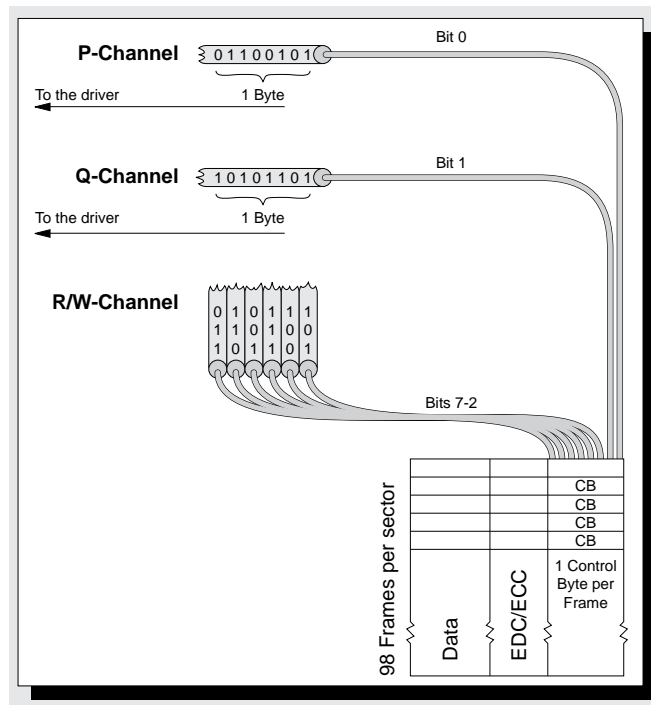
Above we found that each frame contains a byte of control information. This is called the *control* or *subcode byte*. Taken together from all of the frames in a sector, there are a total of 98 subcode bytes.

The individual bits of each subcode byte are identified by the letters P through W. The first bit is named P, the second Q, and so on. A subchannel results by associating all bits in the same position of the subcode bytes in successive frames. The data stream consisting of the first bits of the 98 subcode bytes in a sector is called the P-channel. The sequence of data made up of the second bits is called the Q-channel. The six bits R through W are combined to make a single R-through-W subchannel.

Subchannel P indicates whether music or computer data is found in a sector.

Subchannel Q contains timing information. This may be either the absolute time measured from the start of the CD (ATime) or the relative track time measured from the start of the track (RTime). In the lead-in areas of the disc, 72 bits of the Subchannel Q contain the table of contents, while the remaining 26 bits are used for synchronization and error correction. The R-through-W subchannel contains data for synchronization and error correction.

Illustration of the subchannels



Storage capacity

The storage capacity of a CD-ROM is determined by the number of sectors. CD-ROM storage capacity ranges from 500 Meg to 680 Meg. This depends on the amount of surface area used to press the disc. Originally some of the surface area was left unused because the CD duplicators had problems pressing the outer 5 millimeters of a CD-ROM. As a result, storage capacity was limited to 550 Meg. As manufacturing techniques improved, the full surface area was used yielding the maximum capacity of a CD-ROM can be used with 682 Meg.

Error correction

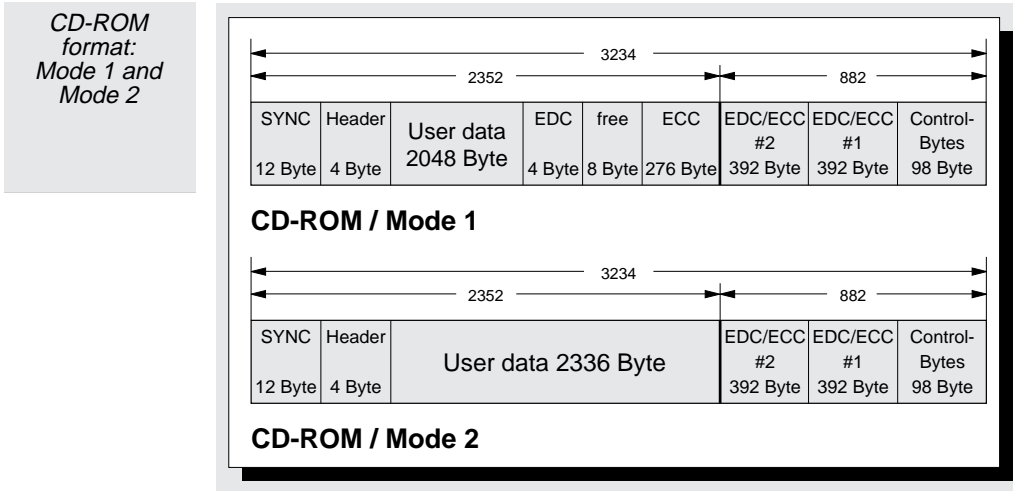
There are 2×392 bytes of error correction information stored in each sector. The contents of this information is defined by an algorithm specified in the Red Book. This proven algorithm is used at hardware level for error free transfer of sectors with all audio CD players and CD-ROM drives. It is based on a widely used technique called the "Reed-Solomon-Code". The code was slightly modified for CD-ROMs and called Cross Interleaved Reed-Solomon Code (or simply, CIRC).

Using CIRC, the error rate is kept to about one in 10^8 bits. In other terms, for every 100 million bits of information read, only one bit will not be detected as an error nor corrected. Someone listening to an audio CD will barely notice such an error due to the rapid sequence of sectors in 1/75 second cycle. However, this error rate is still too high for computer users where any defective bit in the program code or critical data can lead to a disaster. To insure a higher level of error-free data throughput, the format for a CD-ROM is different than the Red Book format.

CD-ROM formats

The format for CD-ROM discs is specified in the Yellow Book. It's based on the Red Book and builds on the standards for audio CDs, creating a standard reference for the storage of computer data on the same size discs used for audio CDs.

The following illustration shows the Yellow Book format differs from the CD-DA format in the amount of data that can be stored in each sector. To decrease the likelihood of errors, the Yellow Book specifies additional error detection and correction information be added to each sector. The amount of usable data is reduced to 2,048 bytes (2K) per sector instead of the 2,352 bytes for the CD-DA format. Most programmer's would agree that 2K is an easier number of bytes to work with.



You'll see from the above illustration that the Yellow Book defines two CD-ROM formats - Mode 1 and Mode 2. Data written in Mode 2 doesn't use the additional error detection and correction information. Mode 2 sectors should be used only for data that isn't critical. An example of this type of data is graphic images that are overlaid on the screen for a short time. Remember the basic error detection and correction methods are still used with Mode 2 data, so the chance of an error are still quite low. But using Mode 2, additional data can be packed into a sector. In practice, very few CD-ROMs are written using Mode 2.

One advantage of Mode 2 sectors is the transfer rate of a drive is increased - even if only slightly. If 75 Mode 1 sectors are read per second, the effective transfer rate is $75 \times 2K = 150K$ per second. With Mode 2 sectors, this increases to $75 \times 2,336 \text{ bytes} / 1,024 \text{ bytes per K} = 171K$ per second. Double, triple and quad speed drives are able to transfer more data. Both sector formats have 12 synchronization bytes at the start of the sector and a 4-byte header. It identifies the sector number and type of sector in Red Book. Mode 0 denotes an empty sector filled exclusively with zeros.

XA format

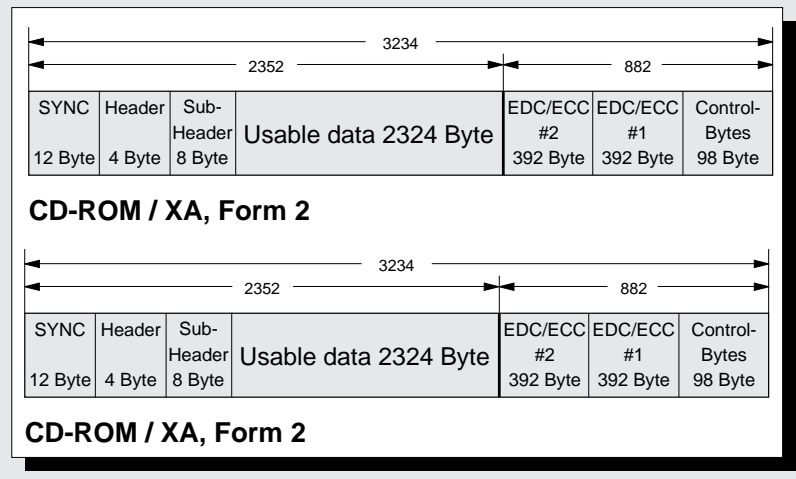
Today's CD-ROM drives also use a format which has become a defacto standard for CD-ROM multimedia titles. This format is called the XA format (also called the CD-ROM XA) developed by Sony and Philips with Microsoft in 1989. A revised XA format, called "eXtended Architecture" (XA) was announced in 1991. Until recently, only few users have been manufactured using this format. CD-ROM/XA is closely connected with CD-I format, which Philips developed for its CD interactive system.

The XA standard was developed to solve a typical problem for multimedia applications. To display text on the screen while a video runs and music plays in the background, an application must simultaneously process three data streams from three different files. Furthermore, the application must do this in real time. However, since only one CPU is available, it must repeatedly work with small amounts of data representing text, audio output and video display. This means the CPU must download a piece of the file with the text, then a piece from the file with the audio and finally a piece from the file with the video in rapid succession. If these files are stored on the CD in different locations, the read head must constantly move from one file to the next. When you consider the slow seek times of a CD-ROM drive, the colorful world of multimedia can quickly collapse. The video becomes jerky, the sound distorted and the text no longer matches the image.

CD-ROM Sector Header, Mode 1 or 2		
Offset	Contents	Type
00h	Sector address: Minute	1 BYTE
01h	Sector address: Second	1 BYTE
02h	Sector address: Sector	1 BYTE
03h	Type of sector 0 = Mode 0 1 = Mode 1 2 = Mode 2	1 BYTE

The important feature of the XA standard is its ability to interleave sectors. By using interleaving, the CPU can read the required text, video and audio without moving the read head. The sectors are nested with the file (or example, three text sectors start a sequence). They're followed by four video sectors and three audio sectors. This repeats itself until all three data streams are completed.

*CD-ROM
format:
Form 1 and
Form 2*



CD-ROM XA has two different formats: Form 1 and Form 2. Both are similar to Mode 1 of CD-ROM format, but differ slightly at the start of the sector, where the usable data is stored.

Both Form 1 and Form 2 use the eight unused bytes of CD-ROM Mode 1 between the data for error detection (EDC) and error correction (ECC). These bytes are shifted with XA to the beginning of the sector where they form a subheader. The subheader contains more information, i.e., information for interleaving, which contrasts XA format from normal CD-ROM format.

Subheader of CD-ROM XA Sector, Form 1 or 2		
Offset	Contents	Type
00h	File number for interleaving 0 = no interleaving	1 BYTE
01h	Channel number for audio data	1 BYTE
02h	XA flag	1 BYTE
03h	Coding flag for audio or video data	1 BYTE *
04h-07h	Duplicate of bytes 00h to 03h	4 BYTES
* 0 with data sectors		

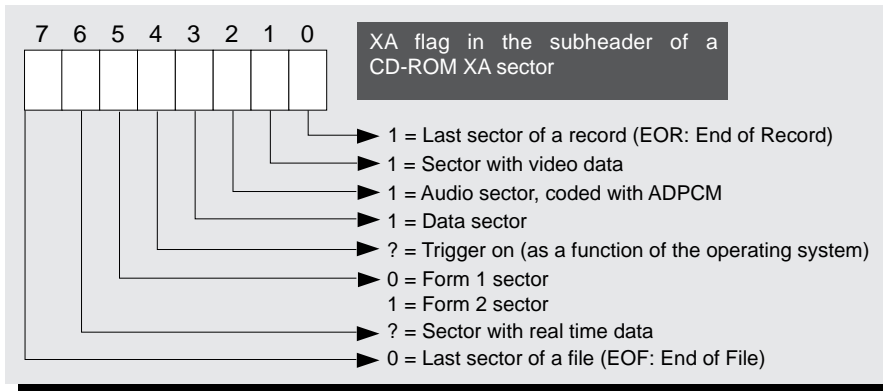
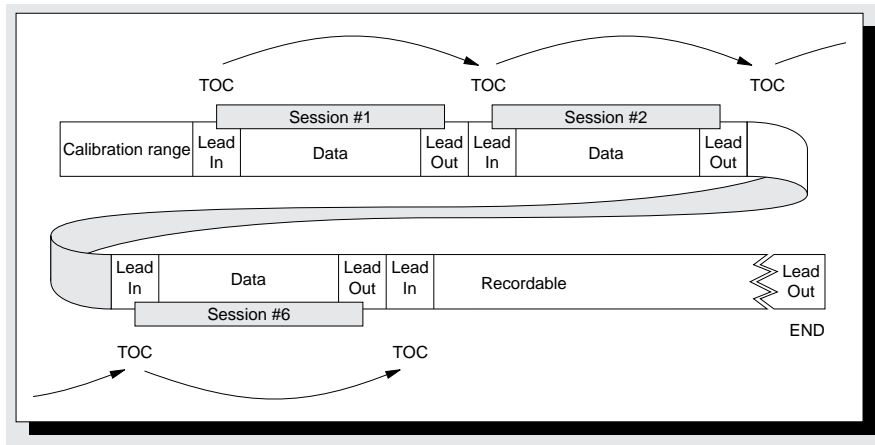


Photo CD and CD-WO

The Photo CD belongs to a class called CD-WO, an acronym for CD-Write Once. You may also hear another term for Photo CDs - term CD-R. CD-R is an acronym for CD-Recordable. The special feature of Photo CDs is that you can write to this type of CD more than once. This standard is described in the Orange Book (the latest book in the "Rainbow Series") published in 1990.

Recordable CDs are immediately recognizable because they're gold colored. In contrast, read-only CDs are silver colored. The reason recordable CDs are gold is to achieve a higher reflection level. Instead of the pits and lands, beneath the gold layer is an organic chemical compound, whose reflective properties can be changed by the laser to simulate pits or lands. This technique also allows standard CD-ROM drives to also read recordable CDs.

*Illustration of a
CD-WO
(CD Write Once)*



Since a CD-WO can be recorded more than once, it must have a different structure than a standard CD-ROM. While the beginning of a standard CD-ROM is identified by a lead-in, the lead-in of a CD-WO is preceded by two other areas used to align the laser along the writable portion of the media.

Each time you record data on a CD-WO you create a session. During a session the lead-in, the data area and the lead-out are written to the media. The table of contents (TOC) to the information in the data area is written to part of the lead-in.

During a following session, the recording after the lead-out of the preceding session. The new volume is written, again composed of a lead-in, a data area and a lead-out. Because the lead-ins are linked by their TOCs, a drive can move from volume to volume (in other words from session to session) gathering the entire contents of the CD-WO.

High Sierra represents the logical format

So far we've talked only about the physical format of the CD-ROM. But for most applications, it's more pertinent to talk about the logical format - files and directories. A peripheral manufacturer can choose to use its own logical format for their CD-ROM drive. But doing so requires proprietary drivers for each operating system. This severely limits the compatibility of CD among users. This is the problem that many in this industry were trying to avoid in 1985 when various representatives of software and hardware manufacturers met to discuss the HSG format. This format is used today for computer CD-ROMs used on PCs and many UNIX systems. The companion CD-ROM to this book follows the HSG format.

The HSG derives its name from the "High Sierra Group". The participants in the development of HSG took the name from their first meeting place, the High Sierra Hotel and Casino in Lake Tahoe, Nevada. A year later this group's released their recommendation, called "Volume and File Structure of Compact Read-Only Optical Disk for Information Interchange". It was standardized by the ISO (International Standards Organization). It has since then been called the ISO Standard 9660, or simply "ISO 9660".

Although the ISO standard accepted 99.5% of the HSG proposal, there are a few slight differences. The most notable difference is in the structure of the directory entries. This is why some talk about HSG format, others talk about ISO-9660 and still others yet talk about HSG/ISO-9660; they all refer to the same standard.

We'll summarize the most important concepts from the ISO specification on the following pages. Anyone who wants to access CD-ROM files from DOS will not need to understand these concepts since CD-ROMs are now standard mass storage devices and are easily accessed just like a hard drive. However, anyone who wants to access the hardware driver of a CD-ROM drive, for example to begin playback of audio tracks, must be familiar with these concepts.

Logical sectors

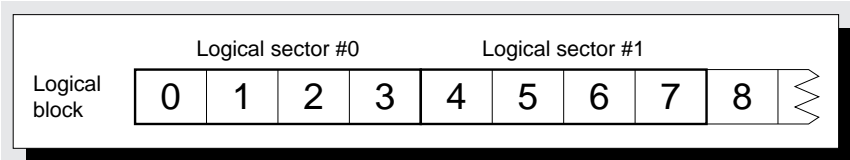
The HSG format defines the logical sector. The size of the HSG logical sector is 2,048 bytes (2K). Each sector has a unique number called a Logical Sector Number (LSN). The first addressable LSN is the number 0 and corresponds to the physical sector at the Red Book address 00:02:00. Therefore, the first 150 physical sectors, which form the first two seconds of a CD, are not accessible at the logical format level. At the same time, this gives us the conversion formula between Red Book addresses (mm:ss:ff) and LSN:

$$\text{LSN}(\text{mm}:\text{ss}:\text{ff}) = (\text{mm} * 60 + \text{ss}) * 75 - 150$$

Logical blocks

To be able to access the components of a logical sector easier and to allow for smaller subdivisions, HSG subdivides a logical sector into several logical blocks. Logical blocks (LBN) can be 512 bytes, 1,024 bytes or 2,048 bytes in size. LBNs are also addressed by numbers. Let's use a logical block size of 512 for this example. Here, LBN 0 represents the first logical block in the first physical sector, 1 represents the second, 2 for the third and 3 for the fourth. Logical block 4 is then at the beginning of the second physical sector.

*Logical sectors
and Logical
blocks*



Files and file names

HSG files are stored as a continuous sequence of logical blocks called an "extent". Therefore, you won't find a FAT (File Allocation Table) on a CD-ROM. By knowing the beginning of a file and its length, you also know all of the LBNs in which the file is stored. Because files on a CD-ROM cannot be deleted, there is no need to fill "freed up" space with fragments of new files (the purpose of a FAT on a hard or floppy drive). Additionally, HSG provides the somewhat exotic option of having files extend over several CDs. This option however, is not supported by DOS.

HSG/ISO-9960 defines the rules for filenames. This is one of the few areas where ISO and HSG differ. HSG filenames follow the Microsoft specifications: 8 character filenames followed by a period followed by a 3 character extension. A filename may contain only uppercase characters A-Z, numerals 0-9 and the underscore "_". ISO filenames are closer to UNIX specifications. A filename can have up to 31 characters with or without a separating period. The filename can be followed by a semicolon, which identifies an optional version number. Due to the long filenames, ISO CDs aren't compatible for DOS users.

Directories and subdirectories

An ISO CD contains a root directory for structuring the stored files. The root directory can have subdirectories. In turn, these subdirectories can have subdirectories, resulting in the tree structure you are familiar with from DOS and UNIX. The only restriction is that maximum number of directory levels is eight.

The root directory and all of its subdirectories are stored as files. The "directory files" can be arranged between all other files anywhere on the CD.

Offset	HSG Field	Type	Meaning	ISO Field	Type
+00h	len_dr	1 BYTE	Length of directory entries in bytes	=	
+01h	XAR_len	1 BYTE	Number of logical sectors reserved for XAR	=	
+02h	loc_extendl	1 DWORD	first logical block of the file in Intel format (Start block number)	=	
+06h	loc_extendM	1 DWORD	ditto in Motorola format	=	
+0Ah	data_lenI	1 DWORD	File length in bytes in Intel format	=	
+0Eh	data_lenM	1 DWORD	ditto in Motorola format	=	
+12h	record_time	6 BYTE	Date and time of the file	record_time	7 BYTE
+18h	file_flags_hsg	1 BYTE	HSG file flags	-	
+19h	Reserved	1 BYTE		file_flags_iso	1 BYTE
+1Ah	il_size	1 BYTE	Number of contiguous sectors for interleave (0=all)	=	
+1Bh	il_skip	1 BYTE	Number of sectors that must be skipped after an interleave block (dividing sectors)	=	
+1Ch	VSSNI	1 WORD	Volume number in Intel format	=	
+1Eh	VSSNM	1 WORD	ditto in Motorola format	=	
+20h	len_fi	1 BYTE	Length of file name	=	
+21h	file_id	n BYTE	File name as a string, length specified in len_fi (up to 32 characters)	=	
+21h+n	padding	1 BYTE	Fill byte, when the next field wouldn't start at an even memory address otherwise	=	
	sys_data	n Byte	Any additional information, from whose length the length of the file name (len_fi), the length of the fixed fields and the total length of the directory entry (len_dr) can be calculated (system data)	=	

You'll note the directory entries for ISO and HSG are identical except for the time field and a connecting flag byte.

Also note that several fields appear twice: once with the suffix I and once with the suffix M. This points to a compatibility problem between Intel and Motorola systems. On Intel systems values larger than 8-bits are arranged low order first, followed by high order. On Motorola systems, the order is reversed. In this table, all 16-bit and 32-bit values appear twice: once in Intel format (I suffix) and once in Motorola format (M suffix). This allows the operating system to use the data with the appropriate format for the processor on which the operating system runs.

On the physical level, the CD-ROM XA sector format manages the requirement for interleaving files. On a logical level, the directory entry manages this task. The two fields `il_size` and `il_skip` indicate how many logical sectors of a file follow one another and how many logical sectors must be skipped until the next block follows with the continuous sectors of the file.

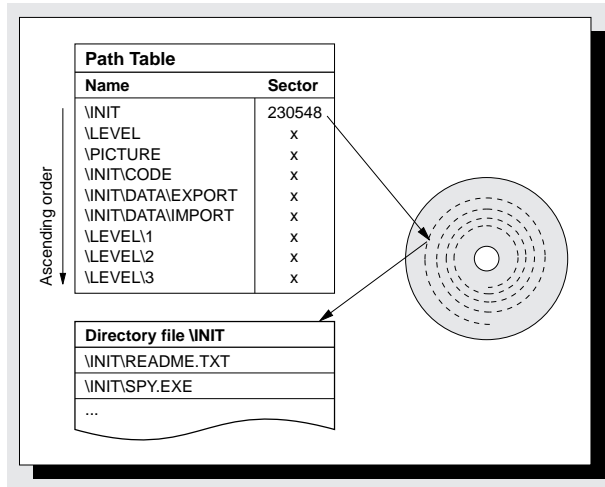
One recommendation is to place no more than 40 files in a directory. By limiting the number of entries in a directory, you'll be able to find a file as quickly as possible. Because so many directory entries fit into a logical sector you'll only need to read the first sector of a directory to find the desired file.

Path table

The standard file system intersperses directory names and filenames. It's a simple and efficient technique but becomes quite involved, especially when searching for files in deeply nested subdirectories. In this case, there may be many directory names to search and read until you find the directory with the desired filename entry. Since the CD-ROM drive has a relatively slow seek times the filename search time increases.

The path table was developed as a way to circumvent the subdirectory problem. The path table contains the names of all the directories and subdirectories on a CD. It also contains the logical sector number at which the subdirectory file begins. By keeping this table in memory, you need only read one sector to determine the address of a file assuming that the directory entry of the file is located in the first sector of the directory data.

Illustration of the path table



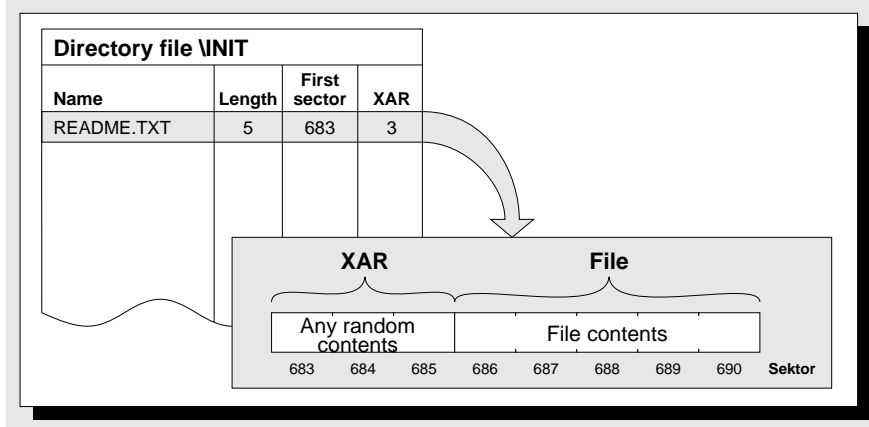
Because the path table contains 32-bit integers representing logical sector numbers, there are two copies of the path table on a CD-ROM. One copy has values in the Intel format and the other in Motorola format.

Extended Attribute Records (XAR)

Another interesting feature is eXtended Attribute Records (XAR). These records give the creator of a file the option of saving as much information about a file as he/she wants, forming the basis of an object-oriented file system. For example, you can keep track of who created the file, whether it expires at a specific time because the data is no longer current and so on.

This information is not saved in the directory entry of a file. Instead it's written to the first logical sector of the file. This prevents the information from needlessly filling up the directory space. The application program or the operating system can then determine how many XAR sectors to skip until the "actual" file begins.

XARs at the beginning of a file

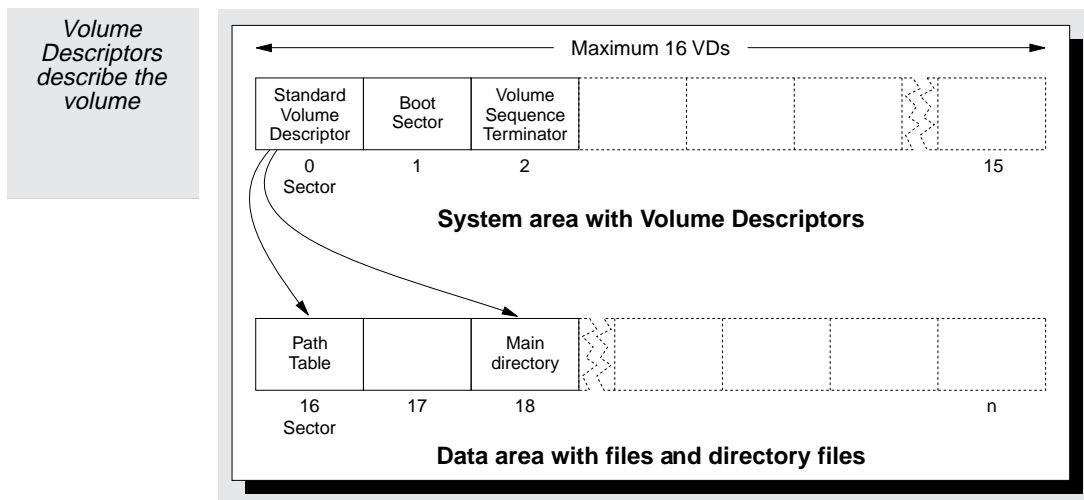


Besides the user definable attributes, HSG also provides some predefined XAR attributes, for example, a user identification, access rights, information about the structure of the records stored in the file and much more. However, these attributes are not applicable under DOS because DOS basically ignores XAR entries.

Volumes

Like the other mass storage systems, another level exists above the level of files and directories on the CD. This level is called the volume. A volume is formed by all the files and directories stored on a CD. HSG describes a volume format based on two components: A system area and a data area. The system area occupies the first 16 logical sectors of a CD (LSN 0 to LSN 15). Its use is not defined and is reserved for the operating system under which the CD is to be used. For example, you could place a boot sector here if you wanted to boot from the CD.

Volume Descriptors (VD) precede the data area of a volume. HSG defines 5 different volume descriptors. Each one describes a different aspect of the medium, but all them fill one complete logical sector. Of the five different VDs, only the "standard volume descriptor" must be present, all the rest are optional. However, you won't find a field with the number of VDs. Instead, a Volume Sequence Terminator indicates the end of the VD.



The most important information contained in a standard volume descriptor is the address of the directory file with the root directory and the address of the path table. The names of the copyright file and the abstract file are also listed. These files are in the root directory. The copyright file gives information about the author of the CD while the abstract file gives information about the CD's contents.

Integrating CD-ROM Drives Into A DOS/Windows Environment

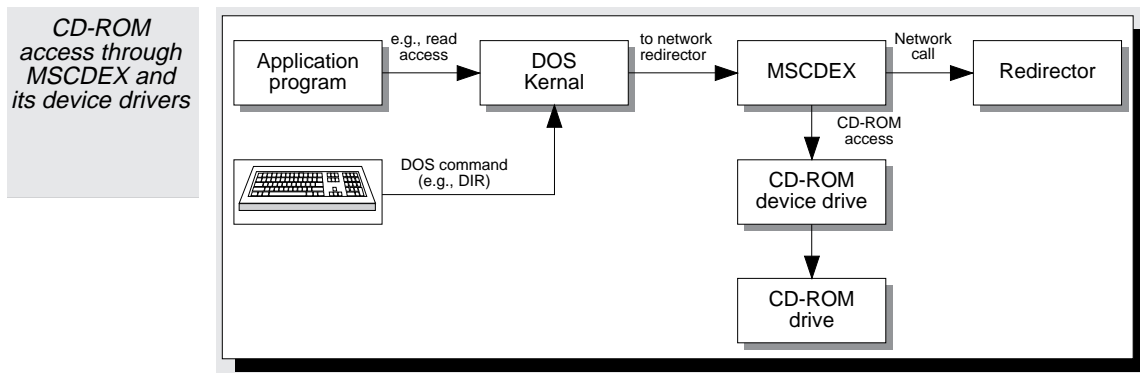
The best hardware is useless if you cannot easily access it from the operating system. DOS uses device drivers which can also be used to access CD-ROM drives under Windows. One problem arose when Microsoft was developing the first drivers for CD-ROM drives under DOS. This new media didn't fit in with the previous concept of device drivers. CD-ROMs are based on the High Sierra format, not on the standard file allocation table format used by DOS. The High Sierra format doesn't recognize the FAT nor any other data structures that DOS uses. At the time the High Sierra format was announced, DOS 3.0 was the current operating system version. DOS 3.0 had a maximum storage capacity of 32 Meg.

This precluded the developers from implementing the CD-ROM driver as a DOS block device. Instead they decided to implement it as a DOS character device. Character drivers follow the same structure as block drivers but cannot be addressed by drive letters like A:, B:, etc., which is the advantage of block drivers. So, another piece of code had to be inserted ahead of the CD driver to make the connection between the DOS kernel and the CD driver. This code is the MSCDEX driver.

MSCDEX

Every PC system which includes a CD-ROM drive also uses the MSCDEX driver. MSCDEX is started from the AUTOEXEC.BAT file. After it is loaded, you can then address the CD-ROM drive using a normal device label (usually D:). Not only is MSCDEX a TSR program but it also attaches itself to the "redirector", a part of the DOS kernel responsible for linking network drives. MSCDEX manages to simulate the existence of a network drive to DOS through the redirector, which in reality is a CD-ROM drive. The only responsibility of the DOS kernel is to provide a device letter to MSCDEX. From then on, this will be the name of the device under DOS. The device letter is usually D: depending on the number of other device drivers and network drives installed in the system.

The DOS kernel handles all accesses to a network drive by delegating them to the redirector. MSCDEX waits there to intercept these accesses. However, MSCDEX doesn't address the CD drive itself, but instead, manages the access through the appropriate device driver. From outward appearances, this is a character driver, but it is a special kind of character driver that performs the functions that MSCDEX requires for access to the drive. More about this later in the chapter.



Besides starting MSCDEX from the AUTOEXEC.BAT file, you have to load the CD-ROM device driver in the CONFIG.SYS file. In fact, you can install several different CD-ROM drivers if you're using CD-ROM drives from various manufacturers. Because a CD-ROM driver depends on the hardware and the interface of the CD-ROM drive, it's usually able to handle multiple drives from "their own" manufacturer, but not drives from other manufacturers.

Until now, the separation of function between MSCDEX and the device driver has worked perfectly: MSCDEX is hardware independent and is isolated from the differences among the various drives by the CD-ROM drivers. This is why manufacturers can supply the same MSCDEX for installation of a drive. However, the actual CD-ROM drivers are hardware dependent. The manufacturer of the drive has to supply the appropriate drivers.

Connecting MSCDEX and the device drivers

One critical area is connecting MSCDEX and the device driver when the PC system is booting. Being part of the AUTOEXEC.BAT file, MSCDEX isn't loaded until after the CD-ROM device drivers are been loaded from the CONFIG.SYS file. Each CD-ROM driver must have a specific name for MSCDEX to recognize it. A name such as MSCD00x (where x represents a number) is often used so that you can add a different driver which can be identified by another name. The complete name is specified by the user when the driver is started from the CONFIG.SYS file. This applies to all drivers whether it be a Sony, NEC, Philips, Hitachi or SoundBlaster CD drive. Although this approach may at first seem a little awkward and complicated, it's necessary to maintain the functional separation between the hardware dependent components and the hardware independent components. Here's an example:

```
device=SB CD.SYS /P:200 /D:MSCD001
```

Programming considerations

After you've installed the drivers and MSCDEX, you can access the CD-ROM drive just like a normal DOS drive. Not only is the drive available at DOS level, where "DIR D:" is enough to display the root directory of the CD-ROM to the screen. The D: drive is now a normal drive on the level of the DOS API (Application Program Interface), and can be used with all the functions for opening and reading files and directories. The corresponding functions of DOS interrupt 21h are available to the programmer without restrictions. However, the nature of the medium prevents all types of write accesses unless you have a CD-ROM recorder.

The unique character of a CD-ROM drive is, to a large extent, hidden from view at the DOS API level. But these unique functions are available. For example, MSCDEX has functions to access the "copyright files" and "abstract files" which are only found on a CD-ROM. But MSCDEX doesn't have a function to playback audio CDs. Because MSCDEX has limited abilities, you'll have to write a device driver to be able to perform all of the operations that are built into the hardware.

Accessing The CD Using The MSCDEX API

MSCDEX is one of the "MUX handlers" (see Chapter 29) which DOS uses with the PRINT, DOSKEY and APPEND commands. MSCDEX also attaches itself to DOS interrupt 2Fh. Calling this interrupt is called with a value of 15h in the AH register is passed onto MSCDEX which then handles the call.

Several versions (1.0 to 1.1, 2.0, 2.1 and 2.2) of MSCDEX are available. The APIs listed in the table to the right are based on Version 2.2 of MSCDEX. A total of 15 functions are defined.

Func. Subr. AX		Task	Version*
AH	AL		
15h	00h	Get number of CD-ROM drives.	1.0
15h	01h	Get CD-ROM driver list	1.0
15h	02h	Get copyright filename	1.0
15h	03h	Get abstract filename	1.0
15h	04h	Get bibliographic filename	1.0
15h	05h	Read volume table of contents	1.0
15h	06h	Reserved	1.0
15h	07h	Reserved	1.0
15h	08h	Read sectors	1.0
15h	09h	Write sectors	1.0
15h	0Ah	Reserved	1.0
15h	0Bh	CD-ROM drive check	2.0
15h	0Ch	Get MSCDEX version number	2.0
15h	0Dh	Get CD-ROM units	2.0
15h	0Eh/ 00h	Get volume descriptor preference	2.0
15h	0Eh/ 01h	Set volume descriptor preference	2.0
15h	0Fh	Read directory entry	2.0
15h	10h	Send device request	2.1

* MSCDEX from Version

Calling a function from the MSCDEX API

When you call one of the MSCDEX function, you'll use the processor registers as follows:

AL	Number of the function to be performed
CX	For functions that expect a drive letter of the CD-ROM drive to be accessed. 0 represents A:, 1 represents B:, etc.
ES:BX	For functions that expect a pointer to a buffer they are passed in this register pair in the form of FAR pointers.
DX	For functions that require a specific number of sectors, this is passed in DX.
DI:SI	For functions that require a sector number, this is passed as a 32 bit integer in the register pair DI:SI. DI contains the high-order 16-bits (the Hi-Word) and SI contains the low-order 16-bits (the Lo-Word).
Carry flag	Indicates an error following a call to one of the MSCDEX functions.
AL	Error code if carry flag is not zero

The complete listing and description of these functions are in Appendix E. These functions are commonly known as the multiplexor functions. The following discussions describe some of the practical uses for the MSCDEX functions.

Checking for a CD-ROM drive

Before calling any MSCDEX function, you should first make certain that MSCDEX is actually installed and its API is available. MSCDEX function 0 is used to check the installation. This function returns the number of CD-ROM drives that MSCDEX is able to address in the BX register.

One way to proceed is to zero the BX register and then call function 0h. If BX still contains zero, then MSCDEX has not been properly installed. If MSCDEX was available, then BX would contain a value of at least one. If not, you can save yourself the trouble of all other MSCDEX calls, because it isn't installed, and most likely a CD-ROM drive is not installed either.

Interrupt 2Fh, MUX Code 15h, Function 00h	Get the number of CD-ROM drives	Version 1.0 and above
---	---------------------------------	-----------------------

Use this function to determine the number of installed CD-ROM drives. You can also use this function to check whether MSCDEX is installed.

Input	AH = 15h AL = 00h	Output	BX = Number of managed CD-ROM drives CX = Drive label for the first CD-ROM drive (0=A:, 1=B:, etc.)
-------	----------------------	--------	--

Remarks:

You should call this function before all other MSCDEX functions to ensure that MSCDEX is installed. To do this, load the value 0 in the BX register before calling the function and check whether BX still contains 0 after the function call. If BX still contains 0, MSCDEX cannot be used since no CD-ROM drive are present.

Although it's possible to determine the drive labels of the CD-ROM drives after the function call from the combination of BX and CX, we recommend using the 0Dh function if an MSCDEX driver in Version 2.0 or above is available. There's no guarantee that consecutive drive labels will be assigned to the installed CD-ROM drives, especially on networks. For example, the first CD-ROM drive might be D:, followed by the network drive E:, with the second CD-ROM drive getting the drive label of F:. Since you can't determine this information from function 00h, use function 0Dh.

Version number

Because the MSCDEX API has changed several times, you should make certain you have a version of MSCDEX that supports the desired function before you call it. Use MSCDEX function 0Ch (introduced in Version 2.0) for this purpose. This function returns the whole number portion of the version number in the BH register, and the decimal portion of the version number in the BL register. For example, BH = 2 and BL = 20 would mean version 2.20.

However, since this function became available with Version 2.0, make certain you don't have Version 1.x of the MSCDEX driver before calling it. To do this, simply load the BX register with a value of 0 before calling the function and then check the register's contents. If the register still contains 0 after the function call, you won't be able to call function 0Ch, because your MSCDEX driver is Version 1.x.

Interrupt 2Fh, MUX Code 15h, Function 0Ch

Query MSCDEX version number

Version 2.0 and above

Call this function to determine the version number of the installed MSCDEX driver.

Input	AH = 15h AL = 0Ch	Output	BH = the whole number portion of the version number (1, 2, etc.) BL = the decimal portion of the version number, e.g., 10 for 1.10 or 2.10
-------	----------------------	--------	---

Remarks:

Call this function before calling function 00h to make certain the MSCDEX is installed. Because this function is not available until MSCDEX Version 2.0, the BX register should be loaded with a value of 0 prior to calling the function. If you still encounter this value after the function call, it means the function could not be executed and you have Version 1.x of an MSCDEX driver.

Determining the drive letters

To access a CD-ROM drive using MSCDEX, you need to know its drive letter. Function 0 returns this information in the CX register. If BX contains 2 and CX contains 3, it means the current system has 2 drives (BX) that can be addressed using drive labels D: (CX) and E:. However, there's a little catch to this equation, especially when network drives are involved. MSCDEX can't always assign consecutive drive labels in such cases, e.g., because E: may already be reserved for a network drive. In the situation we just described, MSCDEX would call the first CD-ROM drive D:, but the second one would not be E:, instead, it would be F:.

Unfortunately, this cannot be handled by function 0 because it is from Version 1.0 of the MSCDEX specification. At the time, developers either didn't recognize this problem or simply ignored it. A newer function, 0Dh, was introduced in Version 2.0 to solve this problem. In addition to the function number in ES:BX, it also expects a pointer to an array that can be a maximum of 26 bytes in size. The function writes the drive labels of the CD-ROM drives addressable using MSCDEX in this array. In the worst case scenario, all 26 drive labels would represent CD-ROM drives. While this is almost impossible, you will know the actual number of drives from a previous call for function 0. Only the first n entries in the array are filled in by the function.

Interrupt 2Fh, MUX Code 15h, Function 0Dh

Get CD-ROM drives

Version 2.0 and above

This function provides the caller with the exact drive labels for the CD-ROM drives managed by MSCDEX. It is preferable to function 00h, because it doesn't assume that all CD-ROM drives will have consecutive drive letters.

Input	AH = 15h AL = 0Dh ES:BX = FAR pointer to a buffer in which the drive labels of the CD-ROM drives are entered.	Output	ES:BX = FAR pointer to buffer with corresponding data
-------	---	--------	---

Remarks:

The buffer must provide one byte for each connected CD-ROM drive. The drive label for the CD-ROM is entered in this byte, with 0 standing for A:, 1 for B:, etc. If the buffer is to be dynamically allocated during program execution, use function 00h to determine the number of CD-ROM drives beforehand, because precisely one buffer entry is required for each CD-ROM drive. On the other hand, if the buffer is statically allocated, you have to figure on the "worst case scenario", providing one entry for each of the maximum 26 CD-ROM drives (from A: to Z:).

If you want to know whether a drive is a CD-ROM drive, simply pass the drive label in the CX-register to MSCDEX function 0Bh. If the drive is actually a CD-ROM drive, a nonzero value will be returned in the AX-register.

Interrupt 2Fh, MUX Code 15h, Function 0Bh

Query CD-ROM drives

Version 2.0 and above

Use this function to determine whether the given drive is a CD-ROM drive being managed by MSCDEX.

Input	AH = 15h AL = 0Bh CX = Drive label for the CD-ROM drive to be accessed (0=A:, 1=B:, etc.)	Output	AX = 0: not a CD-ROM drive <> 0 : is a CD-ROM drive BX = ADADh
--------------	---	---------------	--

Remarks:

After calling the function, you must first inspect the contents of the BX register to determine whether the function call was really handled by MSCDEX and not some other MUX handler. If the BX register contains the MSCDEX signature ADADh, then the contents of AX indicate whether the specified drive is a CD-ROM drive.

Reading and writing

The original task of a drive is to read and write data. Functions 08h and 09h embody these tasks for MSCDEX. The two functions correspond to DOS interrupts 25h and 26h, which are used for absolute reading and writing of sectors from and to storage media. At this level, MSCDEX works with 2K sectors, not files and directories. However, you can only write these sectors if the device is a CD-ROM recorder, which will probably be the case only rarely. To access files and directories, you won't need to use these functions. You can use these functions, for example, when you want to "dump" the contents of specific CD-ROM sectors or copy areas of a CD-ROM.

Interrupt 2Fh, MUX Code 15h, Function 08h	Read Sectors	Version 1.0 and above
--	---------------------	------------------------------

This function corresponds to interrupt 25h which is used to read sectors from any drive. Unlike interrupt 25h, this function can be sent only to a CD-ROM drive.

Input	AH = 15h AL = 08h CX = Drive label for CD-ROM drive to be accessed (0=A:, 1=B:, etc.) SI:DI = Number of the first sector DX = Number of sectors to be read ES:BX = FAR pointer to a buffer for the sectors that are read	Output	Carry flag = 0: OK= 1 : Error, in this case AL = 15: invalid drive specified or 21: drive not ready
--------------	---	---------------	---

Remarks:

Each sector 2,048 bytes, so the buffer must be large enough to accommodate 2,048 bytes per sector.

The number of the first sector to read is coded as a 32-bit value in SI and DI. SI contains the high-order 16-bits (the Hi-Word) while DI contains the low-order 16-bits (the Lo-Word).

Interrupt 2Fh, MUX Code 15h, Function 09h	Write sectors	Version 1.0 and above
--	----------------------	------------------------------

This function can be used only for CD-ROM recorders since a standard CD-ROM drive is read-only. The function corresponds to interrupt 26h, which is used to write sectors to any drive. Unlike interrupt 26h, this function can only be sent to CD-ROM drives.

Input	AH = 15h AL = 08h CX = Drive label for CD-ROM drive to be accessed (0=A:, 1=B:, etc.) SI:DI = Number of the first sector DX = Number of sectors to be read ES:BX = FAR pointer to a buffer for the sectors that are read	Output	Carry flag = 0: OK= 1 : Error, in this case AL = 15: invalid drive specified or 21: drive not ready
--------------	---	---------------	---

Remarks:

The buffer contains 2048 bytes for each sector to be written to the CD-ROM.

The number of the first sector to be written is coded as a 32-bit value in SI and DI. SI contains the high-order 16 bits (the Hi-Word), while DI contains the low-order 16 bits (the Lo-Word).

CD-ROM-specific information

The High-Sierra standard allows three special files to be stored on a CD-ROM. These three files are called the "copyright file", "abstract file" and "bibliographic document file". Although not every CD includes these three files, you can determine if the names of these files are contained on the current CD with functions 02h, 03h and 04h. If these files are present, the function returns the filename, which is always contained in the root directory of the CD. These files are recorded there as a normal file and can be opened and read using the usual functions for accessing files.

Interrupt 2Fh, MUX Code 15h, Function 02h	Get name of copyright file	Version 1.0 and above
---	----------------------------	-----------------------

A CD-ROM may contain a "copyright file" which gives information about the author of the CD. Use this function to determine the name of this file.

Input	AH = 15h AL = 02h CX = Drive label for the CD-ROM drive to be accessed (0=A:, 1=B:, etc.) ES:BX = FAR pointer to the buffer that will contain the filename	Output	Carry flag = 0 OK Carry flag = 1 : Error, in this case AL = 15: invalid drive specified
--------------	---	---------------	---

Remarks:

The buffer should be 38 bytes in length, although the filename of the copyright file for HSG CDs is limited to 8 characters for the name and 3 characters for the extension, thus taking on the conventional form of "nnnnnnnn.eee". The name is written to the buffer as an ASCII string, so is ended by a zero byte. The filename does not contain a path, since the file is assumed to be located in the root directory of the CD-ROM.

If a blank string is returned, the CD-ROM does not contain a copyright file.

Interrupt 2Fh, MUX Code 15h, Function 03h	Get name of abstract file	Version 1.0 and above
---	---------------------------	-----------------------

A CD-ROM may contain an "abstract file" with a brief summary of the CD's contents. Use this function to determine the name of this file.

Input	AH = 15h AL = 03h CX = Drive label for the CD-ROM drive to be accessed (0=A:, 1=B:, etc.) ES:BX = FAR pointer to the buffer that will contain the filename	Output	Carry flag = 0 OK Carry flag = 1 : Error, in this case AL = 15: invalid drive specified
--------------	---	---------------	---

Remarks:

The buffer should be 38 bytes in length, although the filename of the abstract file for HSG CDs is limited to 8 characters for the name and 3 characters for the extension, thus taking on the conventional form of "nnnnnnnn.eee". The name is written to the buffer as an ASCII string, so is ended by a zero byte. The filename does not contain a path, since the file is assumed to be located in the root directory of the CD-ROM.

If a blank string is returned, the CD-ROM does not contain an abstract file.

Interrupt 2Fh, MUX Code 15h, Function 03h	Get name of bibliographic documentation file	Version 1.0 and above
---	--	-----------------------

A CD-ROM may contain a "bibliographic documentation file", which gives information about the sources of the information stored on the CD. Use this function to determine the name of this file.

Input	AH = 15h AL = 03h CX = Drive label for the CD-ROM drive to be accessed (0=A:, 1=B:, etc.) ES:BX = FAR pointer to the buffer that will contain the filename	Output	Carry flag = 0 OK Carry flag = 1 : Error, in this case AL = 15: invalid drive specified
--------------	---	---------------	---

Remarks:

The buffer should be 38 bytes in length, although the filename of the copyright file for HSG CDs is limited to 8 characters for the name and 3 characters for the extension, thus taking on the conventional form of "nnnnnnnn.eee". The name is written to the buffer as an ASCII string, so is ended by a zero byte. The filename does not contain a path, since the file is assumed to be located in the root directory of the CD-ROM.

If a blank string is returned, the CD-ROM does not contain a bibliographic documentation file.

HSG CD-ROM contains a VTOC, or volume table of contents. The VTOC is a type of table of contents with status information about the CD. It fills a complete 2K sector and is read by using MSCDEX function 05h.

Interrupt 2Fh, MUX Code 15h, Function 05h	Read VTOC	Version 1.0 and above
---	-----------	-----------------------

Use this function to read the VTOC of a CD-ROM.

Input	AH = 15h AL = 04h CX = Drive label for CD-ROM drive to be accessed (0=A:, 1=B:, etc.) ES:BX = FAR pointer to the buffer that will contain the filename.	Output	Carry flag = 0 OK Carry flag = 1 : Error, in this case AL = 15: invalid drive specified
--------------	--	---------------	---

Remarks:

The buffer must be 2,048 in length to accommodate the volume table of contents.

Most CD-ROMs contain only one volume, so they also have just one VTOC. Multisession CDs such as Photo CD's or CD-MOs contains multiple VTOCs.

Accessing the CD-ROM using drivers

When MSCDEX functions won't do the trick, you can try switching to the functions of the CD-ROM device driver. Although we'll show how this is done in the next section, you should use the MSCDEX function introduced in Version 2.1 that was created especially for this purpose, function 10h. You only need to pass the function the drive label as well as a pointer to the normal data block which must be present for each call of a DOS driver function. The important advantage to this function is that it uses the specified drive label to find the driver responsible for the drive as well as uses subunit code. Since a CD-ROM driver can manage several drives at the same time, a numerical value in the form of subunit code is passed to the driver to select the desired drive. This is another task that function 10h handles for the caller.

Interrupt 2Fh, MUX Code 15h, Function 10h	Send request to device driver	Version 2.1 and above
---	-------------------------------	-----------------------

Use this function to send a request to the device driver for the CD-ROM drive without having to know the address of the device driver and the subunit code for the CD-ROM drive.

Input	AH = 15h AL = 10h CX = Drive label for CD-ROM drive to be accessed (0=A:, 1=B:, etc.) ES:BX = FAR pointer to buffer with request for device driver	Output	None
--------------	---	---------------	------

Remarks:

Microsoft recommends using this function to communicate with the device driver of a CD-ROM drive because it gives the programmer the option of addressing a device driver without knowing its address and the subunit code of the drive. When

you call this function, MSCDEX automatically puts this information in the device request block which is passed and determines the appropriate device driver from the drive label.

If you want to bypass this call, MSCDEX can return the addresses of the individual drivers and the matching subunit code. Use MSCDEX function 01h to perform this task.

Interrupt 2Fh, MUX Code 15h, Function 01h	Get information about CD-ROM driver	Version 1.0 and above
---	-------------------------------------	-----------------------

Returns information about the various device drivers installed for the CD-ROM drives. Using this information, you can bypass MSCDEX to access the individual device drivers directly.

Input	AH = 15h AL = 01h ES:BX = FAR pointer to buffer for the returned information	Output	None
--------------	--	---------------	------

Remarks:

Each entry in the buffer requires 5 bytes. If the buffer is dynamically allocated during program execution you should determine the number of CD-ROM drives beforehand using function 00h, because each CD-ROM drive requires exactly one buffer entry. On the other hand, if the buffer is statically allocated, you can use the "worst case scenario", and provide an entry for each of the possible 26 CD-ROM drives (from A: to Z:).

The individual entries of the buffer include a FAR pointer to the start of the appropriate device driver and the subunit code. The subunit code is required because a device driver is able of handling several CD-ROM drives, although the drives usually come from the same manufacturer. When you call the driver, you have to specify the subunit code to differentiate between the different drives.

Offset	Contents	Type	
00h	Sub-Unit-Code	1 BYTE	for first CD-ROM drive
01h	FAR pointer to head of appropriate device driver	1 DWORD	
05h	Sub-Unit-Code	1 BYTE	for second CD-ROM drive
06h	FAR pointer to head of appropriate device driver	1 DWORD	
0Ah	Sub-Unit-Code	1 BYTE	for third CD-ROM drive
0Bh	FAR pointer to head of appropriate device driver	1 DWORD	
etc.			

Accessing The CD Using The CD-ROM Driver

MSCDEX doesn't access a CD-ROM drive directly, but instead passes its requests onto the appropriate CD-ROM driver. The following table lists the functions which the device drivers must support under DOS. From a technical viewpoint, a CD-ROM driver is a character driver. In practice it the driver does not have to implement all of these functions.

Driver functions that must be supported by different types of CD-ROM drivers				
Fct	Name/Task	CD-ROM driver	Extended CD-ROM driver	Driver for CD-ROM writer
00h	INIT / Initialization of driver	x		
01h	MEDIA CHECK / Check disk/media change			
02h	BUILD BPB / Create Bios Parameter Block			
03h	IOCTL INPUT / Direct read	x		
04h	INPUT / Read			
05h	NONDESTRUCTIVE INPUT / Read character without removing it			
06h	INPUT STATUS/ Check input status			
07h	INPUT FLUSH / Erase input buffer			
08h	OUTPUT / Write			
09h	OUTPUT WITH VERIFY / Write with verify			
0Ah	OUTPUT STATUS / Check output status			
0Bh	OUTPUT FLUSH / Empty output buffer			x
0Ch	IOCTL OUTPUT / Direct write			
0Dh	DEVICE OPEN / Open device	x		
0Eh	DEVICE CLOSE / Close device	x		
0Fh	REMOVABLE MEDIA / Removable media?	x		
10h	OUTPUT UNTIL BUSY / Output until busy			
17h	GET LOGICAL DEVICE / Get logical device			
18h	GET LOGICAL DEVICE / Set logical device			
80h	READ LONG / Read sectors	x		
81h	Reserved			
82h	READ LONG PREFETCH / Read ahead	x		
83h	SEEK / Find sector	x		
84h	PLAY AUDIO / Play audio		x	
85h	STOP AUDIO / Stop audio		x	
86h	WRITE LONG / Write sectors			x
87h	WRITE LONG VERIFY / Write sectors with verification			x
88h	RESUME AUDIO / Resume audio playback		x	

For a simple CD-ROM driver, the functions that are checked in the first column, "CD-ROM Driver", are sufficient. Several functions with function numbers greater than or equal to 80h fall into this category. These functions have been specially tailored to the requirements of MSCDEX and are unknown to normal device drivers under DOS.

Besides simple CD-ROM drivers that can only access data CDs, DOS also recognizes extended CD-ROM drivers that support additional functions such as playback of audio tracks. Above all, CD-ROM drives that operate with sound cards can playback normal audio CDs. The device driver supports this ability using the appropriate functions.

A third category of functions are those which are used for CD-ROM recorders. Besides the functions of a normal CD-ROM driver, these drivers must also have functions for mastering or cutting a CD. There are only three functions of this type.

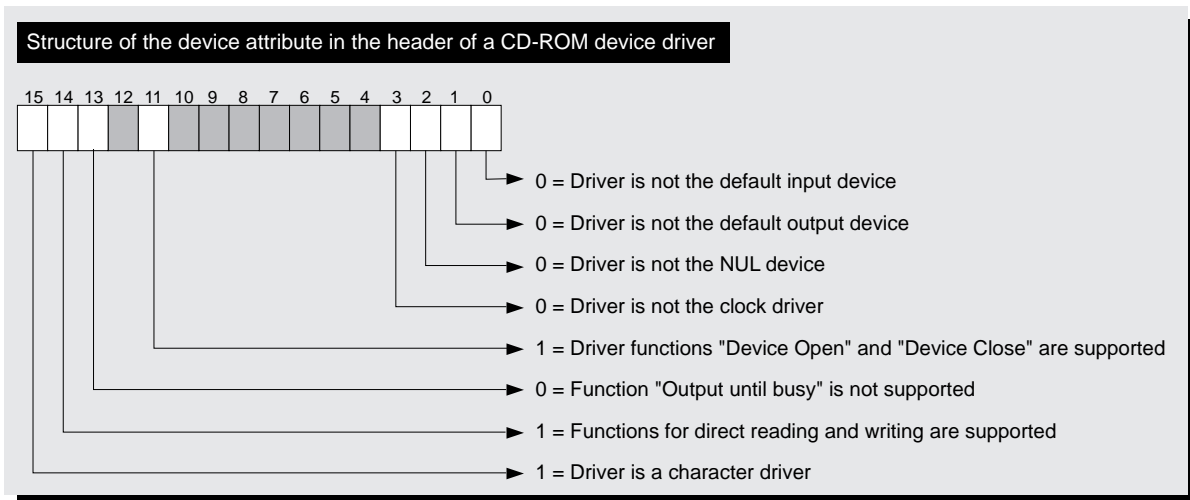
The device header

Like any other DOS device driver, a CD-ROM device driver begins with the device header. This header is located at offset address 0000h of the memory segment into which the driver was loaded. The device header has the normal structure of a DOS device driver except for the two fields at the end of the header. The only unusual feature about the device header is the driver name must read MSCD00x, with x standing for a number. This is the only way MSCDEX can later recognize that the driver is a CD-ROM device driver for which it is responsible.

The header of a CD-ROM device driver		
Address	Contents	Type
+00h	FAR pointer to next driver	1 PTR
+04h	Device attribute	1 WORD
+06h	Offset address of the strategy routine within the driver segment	1 WORD
+08h	Offset address of interrupt routine within driver segment	1 WORD
+0Ah	Driver name (MSCD00x) filled with up to 8 spaces	8 BYTE
+12h	Reserved	1 WORD
+14h	Drive letter for first CD-ROM drive of this driver (0=A:, 1=B: etc.)	1 BYTE
+15h	Number of CD-ROM drives managed by the driver	1 BYTE

The last two fields can be important to a program. You can determine the number of CD-ROM drives managed by the driver and determine the drive label for the first of these drives.

From the device attribute which is stored as part of the device header, it is evident the driver is passing itself off as a character driver to the system. The attribute has the typical structure of device attributes, with the individual flags accepting the following values:



Calling a driver function

All driver functions are available to programs and utilities, except for driver function 00h, which can only be called once when you are loading a driver. Whenever possible, call the driver through MSCDEX function 10h, which is designed especially for this purpose. Function 10h handles various tasks which you would otherwise have to perform yourself if you call the driver function directly:

1. Determines the address of the device header (e.g., using MSCDEX function 01h)
2. Calls the strategy routine of the driver from its address in the device header, passing FAR pointer to ES:BX with the parameters of the function call (the parameter data block).
3. Calls the interrupt routine of the driver from its address in the device header.

Calling MSCDEX function 10h doesn't relieve you of the task of organizing the parameter data block. This structure is your way of communicating with the various driver functions. In this respect, a CD-ROM driver is no different than an ordinary DOS device driver. MSCDEX function 10h also determines the subunit code which tells the driver which of the CD-ROM drives to access within the function call. Programmers who want to call a driver function on their own must enter this information in the parameter data block manually before calling the strategy routine. The information must be entered in the form of a byte at offset address +02h. The structure and size of the parameter data block itself varies depending on function, and must be allocated in memory by the user prior to calling the function.

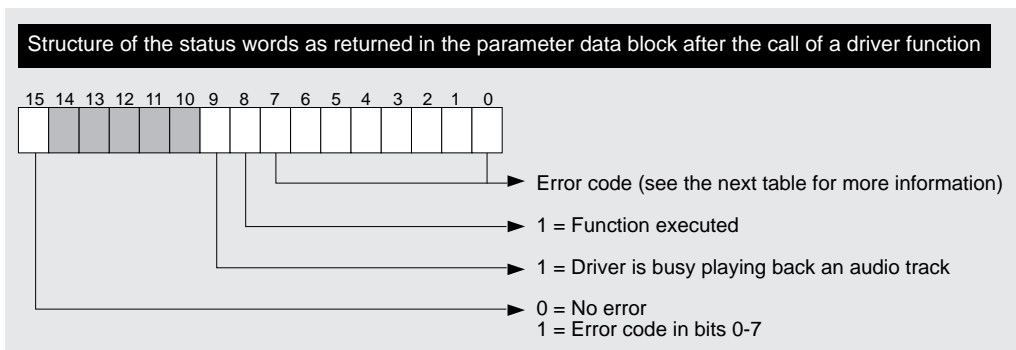
The offsets for the function number and the status word are fixed within the parameter data block. The status word returns information about the success or failure of the function following the call. The function number is always specified at offset address 01h in the parameter block. The status word is always returned at offset 03h of parameter block. If you have worked with other device drivers under DOS, then you'll already be familiar with this, because nothing has changed for CD-ROM drivers. A typical parameter data block will look like the following:

Input and output parameters in the parameter data block when calling function 03h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (03h)	1 BYTE
+02h	Subunit Code for the drive	1 BYTE
+0Eh	Pointer to data structure with additional information	1 PTR
Output parameter		
+03h	Status Word	1 WORD

A program should always begin by calling driver function 0Dh. This function informs the driver that another caller is using one of its functions so it can prepare for subsequent calls. The first function The program should end by calling driver function 0Eh which lets the driver know that it has finished processing. Driver function 0Eh 'closes' the driver to the program.

Status word

The status word tells the user whether the desired function was completed properly. If the highest-order bit is set after the function call, an error has occurred, whose meaning can be taken from the error code in bits 0 to 7.



00h	Write access not possible	07h	Medium type unknown (wrong CD-ROM format)
01h	Unknown drive	08h	Sector not found
02h	Drive not ready (CD not inserted)	0Ah	Write error
03h	Unknown function/function not supported	0Bh	Read error
04h	CRC error	0Ch	General error
05h	Parameter data block wrong length	0Eh	Medium not inserted
06h	Seek error	0Fh	Invalid disk change

When the driver is used for audio playback, bit 9 is used for a special purpose. This is the BUSY bit. When this bit is set, the drive is busy playing an audio track. In this situation, all function calls which result in the movement of the read head are stopped with an error code until the track has finished playing. The BUSY bit is then cleared and the driver is again ready to accept function calls that move the read head.

Querying and setting status information

A frequently called driver function is 03h for direct reading. It returns status information that affect the driver, the CD-ROM drive and the inserted CD. For example, you can query the status of the drive or the current position of the read head. For audio, this function returns information about the number of individual tracks, their length and the total playing time of the CD. You can also use driver function 03h to query the CD's universal product code. This is a unique code of 13 BCD numbers that are also printed on the CD jacket. It is used as a bar code for identifying the CD at scanner cash registers.

Because its unique, some CD player software uses a UPC code. Many CD player software programs can catalog your CD albums by letting you type the title and then automatically saves both the title and UPC code in a database. The next time the user inserts the CD, the CD player software can automatically display the CD's title because it recognizes the UPC code.

Function 0Ch (direct write) is the counterpart of function 03h. Not only can you make various settings to the driver with this function, you can also perform such elementary tasks as ejecting a CD from the drive. This gives a program immediate control over the drive, something you cannot achieve with MSCDEX functions.

Playing audio tracks

An operation that you cannot perform through MSCDEX is to start the playback of audio tracks. Most CD-ROM drives have audio playback capability, particularly the drives bundled with sound cards. Besides the drive interface cable, a three or five pin cable usually connects the audio output of the CD-ROM drive to the audio input line of the sound card. The audio signals are then played through the speakers connected to the sound card.

During audio playback, MSCDEX is not involved. You can control the playback through driver functions 84h, 85h and 88h, which were especially introduced for this purpose. Before the first call you should obtain the drive status using driver function 03h. Then you can determine whether the driver is capable of audio playback and, if so, whether it has the appropriate functions.

CD-ROM device driver functions

The following pages list the 15 different functions that a CD-ROM driver, depending on what type it is and the CD-ROM drive itself, must support.

INIT Initialization (00h)

Like an ordinary driver, the initialization function of a CD-ROM driver is only called once (during initialization of the driver). Even in the case of a CD-ROM driver, it's the DOS kernel, not MSCDEX, that first calls this function. Because the driver must pass itself off as a character driver to DOS, the value 0 is returned for the number of supported devices, as DOS expects from a character driver.

Input and output parameters when calling function 00h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+02h	Function number (00h)	1 BYTE
+0Eh	Maximum end address of the driver (DOS 5.0 and above)	1 PTR
+12h	Address of character that follows the equals sign after the DEVICE command in the CONFIG.SYS file	1 PTR
Output parameters		
+03h	Status Word	1 WORD
+0Dh	Number of supported devices, must be set to 0	1 BYTE
+0Eh	Address of first free byte after the driver	1 PTR
+16h	Drive label, must be set to 0 for MSCDEX	1 BYTE

Besides the normal initialization functions of a device driver, the CD-ROM driver must also place the device name in the device header during execution of the initialization function. MSCDEX, which won't be active until later, expects the name "MSCDEX00x" in the device header so it can recognize the driver as one of its own. The x represents a consecutive number that helps differentiate between the different CD-ROM device drivers in the system. The device drivers take these numbers (or the entire driver name) from the DEVICE= line in the CONFIG.SYS. This is the command line that loads the device drivers into memory. For example, here's a device driver for CD-ROM drives connected to a SoundBlaster card.

```
device=SB CD.SYS /P:220 /D:MSCD001
```

So it's the task of the initialization function to filter this name from the DEVICE= line and place it in the device header, where it must be filled with up to eight spaces.

IOCTL INPUT

Read (03h)

This function returns certain status information. The information reflects different facets of the current drive status. The caller determines which information is to be returned.

Input and output parameters when calling function 03h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (03h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
+0Eh	Pointer to data structure with additional information	1 PTR
Output parameter		
+03h	Status Word	1 WORD

The caller passes a pointer to a data structure at offset address 0Eh in the parameter data block. The driver function infers the type of information that is to be returned from the data structure. The first byte of this data structure contains the code for the desired information. The byte serves as a type of subfunction number for the CD-ROM driver. The CD-ROM driver returns the desired information in the default sequence to the caller following this byte. The size of the buffer is determined by the subfunction and the parameters the subfunction returns.

If the driver does not know the specified subfunction number or it is not supported, it returns error code 3 for "Unknown Command" in the status word. If the driver can return the desired information, but is unable to do so at the time of the call, it returns error code 2 for "Drive Not Ready" in the status word.

The following subfunctions are available.

Subfunctions of function 03h of a CD-ROM driver					
Code	Meaning	Buffer size/BYTE	Code	Meaning	Buffer size/BYTE
0	Get address of device header	5	8	Request size of current volume	5
1	Get position of read head	6	9	Request information on disk change	2
2	Reserved	-	10	Request information on audio CD	7
3	Return error information	-	11	Request information on audio track	7
4	Get information on audio channels	9	12	Request Audio-Q-Channel	11
5	Request drive information	130	13	Request Audio-Sub-Channel	13
6	Request drive status	5	14	Request UPC (Universal Product Code)	11
7	Get sector size	4	15	Request status of audio playback	11

Subfunction 0

Get address of device header

Use this subfunction to determine the address of the device header. Other programs wanting to inspect the device header can also use the subfunction.

Data block for executing subfunction 0 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (00h)	1 BYTE
To be loaded by the subfunction		
+01h	FAR pointer to device header	1 PTR

Subfunction 1

Get position of read head

Use this subfunction to determine the current position of the read head in HSG or Red Book format. Although the device driver is free to choose either format, it must signal the format in the data block of the subfunction using a corresponding flag.

Data block for execution of subfunction 1 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (01h)	1 BYTE
To be loaded by subfunction		
+01h	Format (0 = HSG, 1 = Red Book)	1 BYTE
+02h	Position of read head as logical block number in HSG format or as a unit of minutes/seconds/frames in Red Book format	1 DWORD

Subfunction 3

Return error information

This subfunction is designated for future use. The structure of the returned error information is not defined. As a result, CD-ROM drivers don't have to implement this function.

Data block for executing subfunction 3 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (03h)	1 BYTE
To be loaded by subfunction		
+01h	undetermined array with error information	variable

Subfunction 4 Get information about audio channels

Use this subfunction to request information about how the input channels are assigned to output channels as well as the volume of the output channels.

Data block for executing subfunction 4 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (04h)	1 BYTE
To be loaded by subfunction		
+01h	Input channel (0 - 3) for output channel 0	1 BYTE
+02h	Volume for output channel 0 (0 - 0FFh)	1 BYTE
+03h	Input channel (0 - 3) for output channel 1	1 BYTE
+04h	Volume for output channel 1 (0 - 0FFh)	1 BYTE
+05h	Input channel (0 - 3) for output channel 2	1 BYTE
+06h	Volume for output channel 2 (0 - 0FFh)	1 BYTE
+07h	Input channel (0 - 3) for output channel 3	1 BYTE
+08h	Volume for output channel 3 (0 - 0FFh)	1 BYTE

Subfunction 5 Request drive information

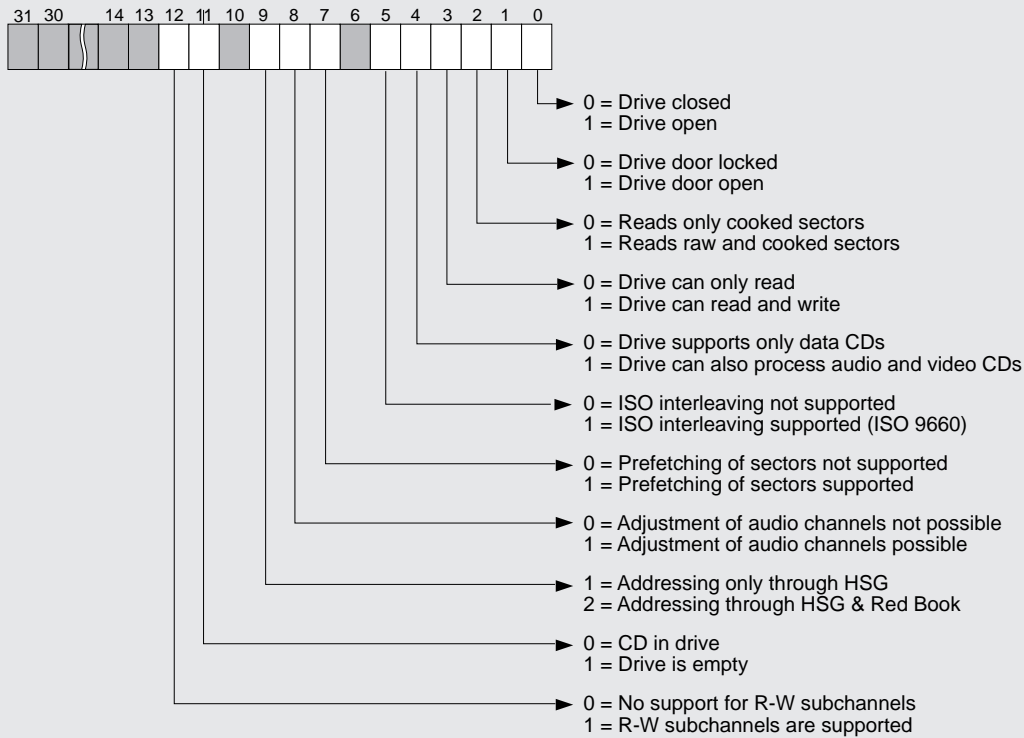
This subfunction is reserved for use by the CD-ROM manufacturer. It allows up to 128 bytes of data to be read from the CD-ROM drive. This forms a type of communications channel between the drive and its firmware. This subfunction is not intended for use by users.

Data block for executing subfunction 5 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (05h)	1 BYTE
To be loaded by subfunction		
+01h	Number of returned bytes (maximum 128)	1 BYTE
+02h	Buffer for transmission of information	max. 128 Byte

Subfunction 6 Request drive status

Use this subfunction to determine the current drive status and the capabilities of the drive.

The drive status as returned by subfunction 6 of the function 03H for a CD-ROM driver



Data block for execution of subfunction 6 of function 03h of a CD-ROM driver.

Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (06h)	1 BYTE
To be loaded by subfunction		
+01h	Drive status	1 DWORD

Subfunction 6 Request drive status

Use this subfunction to determine the sector size, either in raw or cooked mode.

Data block for execution of subfunction 7 of function 03h of a CD-ROM driver.

Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (07h)	1 BYTE
+01h	Read mode (0 = raw, 1 = cooked)	1 BYTE
To be loaded by subfunction		
+02h	Sector size in specified mode in bytes	1 DWORD

Raw sectors take up 2352 bytes on a CD-ROM, and contain correction and control information in addition to usable data. Cooked sectors contain only pure usable data and comprise 2048 bytes (2K).

Subfunction 8

Request size of current volume

Use this subfunction to determine the size of the current volume on the CD. The subfunction gets its information by calculating the corresponding sector number at the position of the Lead-Out-Track, as recorded in the TOC (table of contents). Here is the conversion formula for converting the Red Book start position of the Lead-Out-Track to the HSG sector number:

$$(\text{Minute} * 60 * 75) + (\text{Second} * 75) + \text{Frame}.$$

Data block for execution of subfunction 8 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (08h)	1 BYTE
To be loaded by subfunction		
+01h	Size of volume in sectors	1 DWORD

Subfunction 9

Request information about disk change

Use this subfunction to determine whether the CD in the drive has been changed since the last time you called this subfunction. This subfunction is preferable to driver function 1 (media check) which performs the same task but is not implemented with CD-ROM device drivers.

Data block for execution of subfunction 9 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (09h)	1 BYTE
To be loaded by subfunction		
+01h	Status -1 : CD has been changed 0 : The driver doesn't know whether a change has taken place 1 : CD has not been changed	1 BYTE

Subfunction 10

Request information about Audio CD

This subfunction determines the number of the first and last tracks. This also results in the number of tracks on the CD. The subfunction returns the address of the Lead-Out-Track in Red Book format (representing the total playing time of the CD).

Data block for execution of subfunction 10 of function 03h of a CD-ROM driver		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (0Ah)	1 BYTE
To be loaded by subfunction		
+01h	Number of first track	1 BYTE
+02h	Number of last track	1 BYTE
+03h	Beginning of Lead-Out-Track in Red Book format (playing time)	1 DWORD

Subfunction 11

Request information about the audio track

Use this subfunction to determine the beginning of a track and status information about the track. The caller must also pass the number of the desired track in the buffer before calling the subfunction.

Data block for execution of subfunction 11 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (0Bh)	1 BYTE
+01h	Number of the track	1 BYTE
To be loaded by subfunction		
+02h	Start of track in Red Book format	1 DWORD
+06h	Status information about track (Hi-Nibble CONTROL, Lo-Nibble ADR)	1 BYTE

The following codes apply to the status information, with bit 5 (x) serving as a copy bit, 0 = "digital copy allowed" and 1 = "digital copy not allowed". This code does not effect the ability of a PC to read the corresponding track, storing the information in digital format and then duplicating it. Instead, the code indicates to audio devices whether duplication of the track is allowed.

Status information about an audio track as returned by subfunction 11 of function 03h of a CD-ROM driver		
Code CONTROL	ADR	Meaning
00x0	0000b	Track contains 2 audio channels without Pre-Emphasis
00x1	0000b	2 audio channels with Pre-Emphasis
10x0	0000b	Track contains 4 audio channels without Pre-Emphasis
10x1	0000b	4 audio channels with Pre-Emphasis
01x0	0000b	Not an audio track, but a data track

Subfunction 12

Request audio Q-Sub-Channel

Use this subfunction during playback of audio tracks to get information about the current position of the read head both within the current track and in relation to the entire CD. Status information about the current track is also output. All the information comes from the Q-Sub-Channel of the CD, which is continuously read during playback.

Data block for execution of subfunction 12 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (0Ch)	1 BYTE
To be loaded by subfunction		
+01h	Status information as with subfunction 11 (CONTROL and ADR bytes)	1 BYTE
+02h	Track number as BCD-Value, when ADR = 1, conversion to binary format takes place	1 BYTE
+03h	Current index as BCD-Value (POINT)	1 BYTE
+04h	Minute within the current track	1 BYTE
+05h	Second within the current track	1 BYTE
+06h	Frame within the current track	1 BYTE
+07h	0 (Reserved)	1 BYTE
+08h	Minute within total playing time	1 BYTE
+09h	Second within total playing time	1 BYTE
+0Ah	Frame within total playing time	1 BYTE

Subfunction 13 Request audio subchannel

Use this subfunction to read the R-W subchannels, i.e., the low-order 6 bits from the subchannel of several sequential frames. This results in 96 bytes per sector that contain the subchannel information. Allocation of this subfunction by a CD-ROM device driver is optional and is indicated by bit 12 in the attribute of the device driver.

Data block for execution of subfunction 13 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (0Dh)	1 BYTE
+01h	Number of first frame whose subchannel is to be read in Red Book format	1 DWORD
+05h	FAR pointer to buffer in which the subchannel information is to be placed	1 PTR
+09h	Number of frames to be read (sector number)	1 DWORD

Subfunction 14 Request UPC

Use this subfunction to read the universal product code, a 13 digit number that identifies the CD. This number is also printed on the CD jacket as a bar code (for scanner cash registers). This code is recorded in the Q subchannel of a mode 2 frame.

Data block for execution of subfunction 14 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (0Eh)	1 BYTE
To be loaded by subfunction		
+01h	CONTROL/ADR-Flag	1 BYTE
+02h	UPC/EAN-Code	7 BYTE
+09h	0	1 BYTE
+0Ah	Reserved	1 BYTE

A value of 0 is returned in the CONTROL and ADR bytes if the CD does not contain a UPC. Otherwise, the UPC is present as a BCD number with each of the 7 bytes containing two BCD digits. The last digit, i.e., the low-order nibble in the seventh byte, is always 0.

Subfunction 15

Request status of audio playback

Use this subfunction during playback of audio CDs, to determine the current status of the playback. Among this information is a pause flag and the start and end position for the next RESUME command.

Data block for execution of subfunction 15 of function 03h of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (0Fh)	1 BYTE
To be loaded by subfunction		
+01h	Audio status with bit 0 as pause flag Bit 0 = 0 : Pause not active 1 : Pause active	
+03h	Start position for next RESUME command as Red Book address	1 DWORD
+07h	End position for next RESUME command as Red Book address	1 DWORD

INPUT FLUSH

Erase input buffer (07h)

Use this function to erase all its input buffers, cancel any requests not yet processed and any driver calls.

Input and output parameters when calling function 07h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (07h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

OUTPUT FLUSH

Empty output buffer (0Bh)

Use this function to write all unwritten output buffers to the CD. This driver must refer to a CD-ROM recorder.

Input and output parameters when calling function 07h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (07h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

IOCTL OUTPUT

Write (0Ch)

This function lets you specify various settings for CD-ROM device driver. For example, you can open and close the drive door and eject the CD.

Input and output parameters when calling function 0Bh of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (0Bh)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

Like the Read driver function, the caller must pass a pointer to a data structure at offset address 0Eh in the parameter data block. The driver function determines the type of setting to be made from this data structure. The first byte of this data structure must contain the code for the desired settings, as a type of subfunction number. Depending on the subfunction, the caller must add various additional information to this byte so the parameter data block describes the function call completely.

The table on the right lists the available subfunctions.

Subfunctions of function 0Ch of a CD-ROM device driver		
Code	Meaning	Buffer size/BYTE
0	Eject CD-ROM	1
1	Lock/unlock drive door	2
2	Reset drive	1
3	Set audio channels	9
4	Send control information directly to drive	variable
5	Close door	1

Subfunction 0 Eject CD-ROM

Use this subfunction to eject the inserted CD, unlocking the drive door at the same time.

Data block for execution of subfunction 0 of function 0Ch of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (00h)	1 BYTE

Subfunction 1 Lock/unlock drive door

Use this function to lock or unlock the drive door. Locking the door is required for playing the CD and unlocking it is required for removing the CD. Not all drives are physically able to lock the door. Nevertheless, this subfunction must also be called with these drives.

Data block for execution of subfunction 2 of function 0Ch of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (02h)	1 BYTE

Subfunction 2 Reset drive

Use this subfunction after an error to reset and reinitialize the drive.

Data block for execution of subfunction 1 of function 0Ch of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (01h)	1 BYTE
+01h	0 = unlock, 1 = lock	1 BYTE

Subfunction 3 Set audio channels

Use this subfunction to set the audio channels. You can set the current assignment between the four input and output channels as well as the volume of the output channels.

Data block for execution of subfunction 3 of function 0Ch of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (03h)	1 BYTE
+01h	Input channel (0 - 3) for output channel 0	1 BYTE
+02h	Volume for output channel 0 (0 - 0FFh)	1 BYTE
+03h	Input channel (0 - 3) for output channel 1	1 BYTE
+04h	Volume for output channel 1 (0 - 0FFh)	1 BYTE
+05h	Input channel (0 - 3) for output channel 2	1 BYTE
+06h	Volume for output channel 2 (0 - 0FFh)	1 BYTE
+07h	Input channel (0 - 3) for output channel 3	1 BYTE
+08h	Volume for output channel 3 (0 - 0FFh)	1 BYTE

Not all drives are capable of setting the volume of the various channels to the precision in the range of 0 to 255. With these drives, the driver has the task of reproducing the specified value in one of the drive's possible settings. In the simplest case 0 means channel off and 255 means "maximum volume".

Channel 0 represents the left output and 1 represents the right output, while 2 and 3 stand for the two "Prime outputs" which make quadrophonic sound possible. Output channel 2 represents the left prime channel, while 3 represents the right channel. Drivers whose drive only recognizes the two normal stereo outputs simply ignore the settings for channels 2 and 3.

Subfunction 4 Send control information directly to the drive

This subfunction is reserved for communication between the firmware of a CD-ROM drive (utilities, for example) and the drive. The subfunction creates a communications channel through which the firmware can directly control the drive.

Data block for execution of subfunction 4 of function 0Ch of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (04h)	1 BYTE
+01h	Any data	variable

Subfunction 5 Close door

Use this subfunction to close the door automatically on a drive with that capability.

Data block for execution of subfunction 5 of function 0Ch of a CD-ROM driver.		
Address	Contents	Type
To be loaded by caller		
+00h	Subfunction number (05h)	1 BYTE

DEVICE OPEN Open device (0Dh)

Use this function to notify the CD-ROM driver that it intends to use CD-ROM driver functions.

Input and output parameters when calling function 0Dh of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (0Dh)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

DEVICE CLOSE Close device (0Eh)

Use this function to notify the CD-ROM driver that it has completed using the CD-ROM driver functions.

Input and output parameters when calling function 0Eh of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (0Eh)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

READ LONG Read sectors (80h)

Use this function to read sectors of the CD directly from the CD-ROM device driver, with both Red Book addressing and HSG being supported. You can request the sectors as cooked sectors with the pure 2048 bytes of usable data, or as raw sectors. If you request raw sectors, 2352 bytes must be reserved for each sector in the buffer. This is because the driver function will then copy the sync and header bytes and the EDC bytes to the buffer in addition to the user data. If a driver cannot return part of the control data, it pads the corresponding portion of the sector contents with NULL bytes so the rest of the sector layout does not shift.

Input and output parameters when calling function 80h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (80h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
+0Dh	Addressing mode (0 = HSG, 1 = Red Book)	1 BYTE
+0Eh	FAR pointer to buffer for receiving read sectors	1 PTR
+12h	Number of sectors to be read	1 WORD
+14h	First sector to be read	1 DWORD
+18h	Read mode (0 = Cooked (2048 bytes), 1 = Raw (2352 bytes))	1 BYTE
+19h	Interleave Size	1 BYTE
+1Ah	Interleave Skip	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

The two interleave parameters are specified only if the driver supports reading in interleave mode. In this case, Interleave Size represents the number of contiguous sectors and Interleave Skip represents the number of sectors that have to be skipped to get to the next part of the file.

READ LONG PREFETCH Read ahead (82h)

This function was introduced to increase the performance of CD-ROM drives. Operating systems use this function (in our case, the operating system is MSCDEX) to have the driver read predetermined sectors that will probably be needed soon. For example, it makes sense to use Read Long Prefetch when a large directory occupies contiguous sectors or when a program reads a file from CD-ROM step by step. In such situations, MSCDEX is able to guess which sectors will be needed soon.

Before MSCDEX passes the read sectors on to the operating system or program, it can use this driver function to read the next sectors. This is a method of getting around the relatively slow access times of a CD-ROM drive, because when the guess turns out to be right, the driver will have already placed the desired sectors in a type of internal cache and can return them immediately. At the very least, the driver will have performed a seek on the specified sector so that access can take place more quickly.

Input and output parameters when calling function 82h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (82h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
+0Dh	Addressing mode (0 = HSG, 1 = Red Book)	1 BYTE
+12h	Number of sectors to be read	1 WORD
+14h	First sector to be read	1 DWORD
+18h	Read mode (0 = Cooked (2048 bytes), 1 = Raw (2352 bytes))	1 BYTE
+19h	Interleave Size	1 BYTE
+1Ah	Interleave Skip	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

The data to be passed during the call for the driver is identical to the data of driver function 80h, except for the missing buffer address. A buffer address isn't required, since this function returns no data. Instead it's a notification to the driver that a read operation is imminent. The driver will relinquish control to the caller immediately. The driver can then prepare for the expected read access in the background while the user is performing other programming tasks.

SEEK Find sector (83h)

Use this function to move the read head of the CD-ROM drive to the specified sector thereby eliminating the seek time for a subsequent read access to this sector. The driver will relinquish control to the caller immediately. The driver can then perform the seek to the desired sector in the background while the user is performing other programming tasks.

Input and output parameters when calling function 83h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (83h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
+0Dh	Addressing mode (0 = HSG, 1 = Red Book)	1 BYTE
+14h	Sought sector	1 DWORD
Output parameter		
+03h	Status Word	1 WORD

PLAY AUDIO Play audio (84h)

Use this function to start the playback of an audio track. The function doesn't operate at track level, but rather at sector level, so the address of a track must be determined beforehand. You use subfunction 11 of driver function 3 (Read) for this purpose. By specifying a sector address, playback can also be started in the middle of the track.

The driver returns to the caller immediately after playback starts and continues playback in the background until the specified ending address is reached. The function uses the Busy bit in the status word to indicate that the driver is in play mode and therefore cannot accept any read or seek operations. A driver that doesn't support playback of audio CDs simply ignores this function call and doesn't set the Busy bit.

If a program wants to wait until the driver finishes playback, it is advisable to use permanent polling of the audio status using subfunction 15 of driver function 3 (Read). Subfunction 15 returns the corresponding information.

Input and output parameters when calling function 84h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (84h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
+0Dh	Addressing mode (0 = HSG, 1 = Red Book)	1 BYTE
+0Eh	First sector to be played	1 DWORD
+12h	Number of sectors to be played	1 DWORD
Output parameter		
+03h	Status Word	1 WORD

STOP AUDIO Stop audio (85h)

Use this function to stop audio playback. If this function is called while the drive is in pause mode, only the start and end positions for the next RESUME command are reset (see driver function 3, subfunction 15).

Input and output parameters when calling function 85h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (85h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

WRITE LONG Write sectors (86h)

Use this function to write individual sectors to a CD-ROM recorder. This function allows for various sector format, although the specification requires that the drive support only Mode 1. The other formats are optional.

Input and output parameters when calling function 86h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (86h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
+0Dh	Addressing mode (0 = HSG, 1 = Red Book)	1 BYTE
+0Eh	FAR pointer to buffer with sectors	1 PTR
+12h	Number of sectors to be written	1 WORD
+14h	First sector to be written	1 DWORD
+18h	Write mode (0 = Mode 0 1 = Mode 1 2 = Mode 2 Form 1 3 = Mode 2 Form 2)	1 BYTE
+19h	Interleave Size	1 BYTE
+1Ah	Interleave Skip	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

Depending on the write mode, the data to be written has varying lengths. In Mode 0, the buffer is ignored since the sectors will be filled with null values. In Mode 1 and Mode 2 Form 1 the buffer contains 2,048 bytes of data per sector written. In Mode 2 Form 2 the buffer contains 2,336 bytes of data per sector.

WRITE LONG VERIFY Write sectors with verification (87h)

Use this function to write and verify individual sectors to a CD-ROM recorder. The parameters for this function are identical to Write Long (86h) driver function except that the data is verified immediately following the write operation. The results of the verification are returned in the status word. A write error leaves an appropriate error message in the status word.

RESUME AUDIO Resume audio playback(88h)

Use this function to resume playback after playback has been stopped by function 85h.

Input and output parameters when calling function 88h of a CD-ROM driver		
Address	Contents	Type
Input parameters		
+01h	Function number (88h)	1 BYTE
+02h	Subunit code for the drive	1 BYTE
Output parameter		
+03h	Status Word	1 WORD

Sample Programs And CD-ROM Functions

So you can see some of these functions in action, we've written a module named CDUTIL.C which contains wrapper functions for all MSCDEX and driver functions. CDUTIL includes the CDUTIL.H header file, in which all constants and types required for working with the functions of CDUTIL.C are declared. The following tables give you an overview of the available functions, arranged by task area.

Functions for converting between the various addressing modes	
Function	Task
Time2Frame	Converts time to number of frames
Time2HSG	Converts time to HSG address
Time2REDBOOK	Converts time to REDBOOK address
FrameTime	Splits number of frames into minute, second, frame
REDBOOK2Time	Splits REDBOOK address into minute, second, frame
HSG2Time	Splits HSG address into minute, second, frame
REDBOOK2HSG	Convert REDBOOK address to HSG address
HSG2REDBOOK	Convert HSG address to REDBOOK address

Functions for calling MSCDEX functions	
Function	Task
MSCDEX_GetNumberOfDriveLetters	Get number of supported drives
MSCDEX_Installed	MSCDEX installation check
MSCDEX_GetCDRomDriveDeviceList	Get information about drive letters
MSCDEX_GetCopyrightFilename	Get name of copyright file
MSCDEX_GetAbstractFilename	Get name of abstract file
MSCDEX_GetBibDocFilename	Get name of BibDoc file
MSCDEX_ReadVTOC	Read Volume Table of Contents (VTOC)
MSCDEX_AbsoluteRead	Read data from sector(s)
MSCDEX_AbsoluteWrite	Write data to sector(s)
MSCDEX200_CDRomDriveCheck	Is drive a CD-ROM drive?
MSCDEX200_GetVersion	Get MSCDEX version
MSCDEX200_GetCDromDriveLetters	Get all CD-ROM drive letters
MSCDEX200_GetVDPPreference	Get Volume Descriptor Preference
MSCDEX200_SetVDPPreference	Set Volume Descriptor Preference

Functions for calling MSCDEX functions (continued)	
Function	Task
MSCDEX200_GetDirectoryEntry	Read directory entry of a file
MSCDEX210_SendDeviceRequest	Send device request to device, Version 2.1 and above
MSCDEX_SendDeviceRequest	Send device request to device prior to Version 2.1
MSCDEX_ReadWriteReq	Run MSCDEX-IOCTLI_READ/WRITE request
_CallStrategy	Send device request to device driver

Functions for calling driver functions	
Function	Task
cd_IsError	Does status word describe an error?
cd_GetReqHdrError	Get error code
cd_IsReqHdrBusy	Device still busy?
cd_IsReqHdrDone	Last device command completely processed?
cd_GetDeviceHeaderAddress	Get address of device header
cd_GetLocationOfHead	Get location of (read)/write head
cd_GetAudioChannelInfo	Get assignment of input channels to output channels
cd_ReadDriveBytes	Read drive bytes
cd_GetDevStat	Read device status (drive status)
cd_GetSectorSize	Get size of a sector
cd_GetVolumeSize	Get number of sectors of a CD
cd_GetMediaChanged	CD changed?
cd_GetAudioDiskInfo	Get number of titles (songs, tracks)
cd_GetAudioTrackInfo	Get information about a title
cd_IsDataTrack	Is Track an audio track or a data track?
cd_GetTrackLen	Get size of a track.
cd_QueryAudioChannel	Query time information from Q-Sub-Channel
cd_GetAudioSubChannelInfo	Get information about audio subchannels
cd_GetUPCode	Get universal product code (UPC)
cd_GetAudioStatusInfo	Get current audio status
cd_Eject	Open tray
cd_CloseTray	Close tray
cd_LockDoor	Lock/unlock door
cd_Reset	Reset drive
cd_SetAudioChannelInfo	Set assignment between CD channels and output channels
cd_WriteDriveBytes	Writes drive data
cd_PlayAudio	Play audio track
cd_StopAudio	Stop audio playback
cd_ResumeAudio	Resume audio playback
cd_Seek	Find sector

Functions for calling driver functions (continued)	
Function	Task
cd_ReadLong	Read sectors
cd_ReadLongPrefetch	Prepare to read sectors
cd_WriteLong	Write sectors
cd_WriteLongVerify	Write with verify
cd_FastForward	Fast forward current music output 2 seconds

Functions for outputting information to screen	
Function	Task
cd_PrintDiskTracks	Display all titles and playing times
cd_PrintActPlay	Display playing times and title of current track
cd_PrintSektor	Dump sector contents onto screen
cd_PrintDirEntry	Decode directory entry
cd_PrintDevStat	Output device status
cd_PrintUPCode	Output UPC as BCD number

We recommend that you use CDROM.C as an example of working with functions from CDUTIL.C. CDROM is a program that lets you display diverse information about CDs. With CDROM you can output the number of tracks and their duration, the directory entries of the files and the contents of sectors. Also, you can also use this utility to play audio tracks (if your CD-ROM drive has this capability).

You can set all the functions from the command line when you call the program, for example

```
CDROM D: -UPC
```

displays the universal product code of the inserted CD. If you call the program without any switches/parameters, it displays a listing of the different options on the screen.



Sound Blaster And Compatible Cards

Early multimedia was synonymous with sound cards. Today's multimedia is much more, but sound cards remain a very important part of the equation. One of the main reasons for popularity of sound cards in the late 1980s were computer games. These games removed the "business machine" stigma attached to the PC. Today, sound cards are everywhere.

There are many different makes and style of sound cards. However, as with other classes of peripherals, sound cards standards quickly evolved so programmers weren't forced to write for an unlimited number of different sound cards. The name for this standard is the same as the name of the most popular sound card: Sound Blaster. Creative Labs, the manufacturer of the Sound Blaster, developed a card with good features, ease of installation and reasonable priced. By dominating the early sound card market their card became the standard.

Most sound cards today are "Sound Blaster compatible". This chapter discusses these cards.

The Family Of Sound Blaster Cards

Sound Blaster cards from a few years ago are unlike the cards of today. Like most peripherals, new features, improvements and upgrades have been added to the Sound Blaster cards. These features cover three major functional areas:

- Generating synthetic sound. This task is performed by the Yamaha OPL chip found in Sound Blaster cards, and many other compatibles. This chip produces up to 11 voices which you can play back simultaneously. With skillful programming, this chip can coax a veritable fireworks of sound from your PC.

There have been four versions of the OPL chip: Version 1, 2, 3, and finally Version 4. Version 2 that is still considered to be the standard for Sound Blaster compatibility.

- Sampling sound. This task is performed by a "Digital Sound Processor" (DSP) found in Sound Blaster cards. The DSP is manufactured by Creative Labs and also licensed to other manufacturers. In sampling, analog signals from a CD, stereo equipment or a microphone are recorded and saved onto a storage device for later playback.
- Mixing sound. A mixer controls the volume of the various sound sources and combines them.

Blaster environment variable

Regardless of the different capabilities of the various cards in the Sound Blaster family, all Sound Blaster cards use the "Blaster environment variable", which is defined in the AUTOEXEC.BAT file. If you have a Sound Blaster card installed in your computer, there's a command line in your AUTOEXEC.BAT file that look similar to this:

```
SET BLASTER=A220 I5 D1 H5 P330 T6
```

This environment variable contains all the important information about the port address of the installed sound card, the interrupt being used, the DMA channel and more. Programs that access the Sound Blaster card usually query this environment variable so they can determine the parameters for accessing the sound card. The components of this environment variable specify the following information:

Parameters of the Blaster environment variable	
Parameter	Meaning
A	Base port address of the card
I	Interrupt number used
D	8-bit DMA channel
H	16-bit DMA channel, for cards supporting 16-bit DMA transfer
M	Base port address of Sound Blaster Mixer
P	MIDI port
T	Sound Blaster card ID (1 = early versions of Sound Blaster, 3 = Sound Blaster 2.0, 6 = Sound Blaster 16)

One piece of information the variable does not disclose is the exact type of Sound Blaster card installed. However, by knowing the base address, it's relatively easy to determine this information, as we'll see shortly. As you evaluate the environment variable, keep in mind the sequence of parameters may change. The A and I parameters are mandatory. The sample programs in later sections contain routines that analyze the environment variable.

This chapter concerns the most important parts of Sound Blaster cards: synthetic sound generation using FM synthesis, sampling using the digital sound processor of the card and controlling the mixer and speech output. We've omitted MIDI which is available on only a few Sound Blaster cards. Creative Labs does provide library routines for performing many of the tasks. However, most of the examples here involve direct programming of the card and introduce you to some custom libraries. While there is still a certain compatibility between cards from different manufacturers on a hardware level, there is no such thing on a driver level. Since there is no Sound Blaster standard on this level, we have given preference to direct programming.

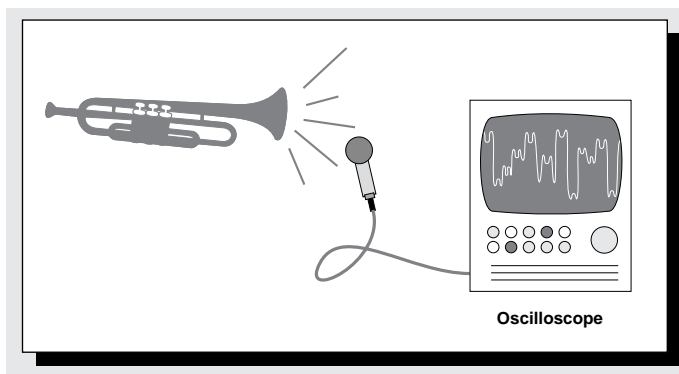
FM Synthesis

Frequency modulation, also called FM or FM synthesis, is the most common form of synthetic sound production using PC sound cards. Like all procedures for synthetic sound production, the goal of FM synthesis is to reproduce the signal progression of natural voices and instruments as true to the original as possible. Therefore, to understand this procedure, you must first look closely at the physical nature of sound.

The physical nature of sounds and frequency modulation

We can "see" sounds generated by a guitar, a piano or a trumpet by using an oscilloscope. The oscilloscope displays sounds as oscillations or vibrations whose path follows a type of sine curve. Such a sine curve is characterized by constant highs and lows over time. It begins with a continuous vibration of the sound carrier, for example a guitar string as it is struck. The string vibrates depending on how hard it is plucked by the player. However, it always vibrates at its characteristic speed, or "frequency". It's this frequency which determines the pitch. For example, the concert pitch of A has a frequency of 400 Hertz (1 Hertz = 1 oscillation per second).

A sound can be recorded electronically using an oscilloscope



The frequency produced is predetermined by the tautness, or tension of the string. The guitar player can modify this frequency by turning the guitar peg to tighten or loosen the string. Anyone who has ever held a guitar in their hands is familiar with this effect. The more you tighten the string, the higher the pitch it generates because the string vibrates more frequently. With an electronic recording, this effect is expressed by a higher frequency, i.e., a faster sequence of highs and lows in the sine curve.

However, frequency is not the only factor which determines the appearance of the sine curve. The force with which the string is struck determines the intensity of the highs and lows of the sine curve. The force determines the amplitude of the vibration. Frequency and amplitude, then, are the deciding attributes of a sound. Therefore, from a mathematical standpoint, you can represent the frequency rate of a sound with the following formula (where $\pi = 3.14$):

$$f(t) = \text{Amplitude} * \text{Sine}(\text{Frequency} * t * 2 \pi)$$

with t representing the time in s (seconds). The factor 2π must be used because the sine function is a radian measure. This requires conversion between the argument t and the radian measure. Because a sine function always returns a value between 1 and -1, the amplitude in this formula provides the maximum and minimum value of the function result.

Sound production is, of course, more complicated in practice than in theory. The sound pattern of a natural instrument is much more complex because several sine waves usually overlap. Moreover, the amplitude doesn't remain constant during the progression of the sound, but instead, builds up first, only to slowly decrease, which the listener perceives as the sound fading away. Still, it is true that you can produce sounds using a simple sine generator, even if the sounds are relatively simple. Daily life is full of such sounds. The dial tone of the telephone, the buzzing of an alarm clock or the beeping of the microwave oven are just a few examples.

Although a simple sine generator can't imitate the complex sound pattern of a guitar or a trumpet, such sine generators are the only resources available to the Sound Blaster and other sound cards. Sine generators are relatively easy to implement and can be found in other areas inside a PC, for example in the timer chip and in the battery-backed realtime clock.

The principal of frequency modulation

Frequency modulation is a technique for creating complex sound patterns. Intuitively, you might think the output of several sine generators are simply combined to make the sounds. However in practice, the sine generators are arranged so the output signal of one generator also becomes the input signal of a second generator. The first generator is termed the modulator, while the second one is called the carrier.

Here's what it looks like mathematically:

$$f(t) = \text{Amplitude2} * \text{Sin}(\text{Frequency2} * t * 2 \pi + (\text{Amplitude1} * \text{Sin}(\text{Frequency1} * t * 2 \pi)))$$

$\pi = 3.14 \dots$

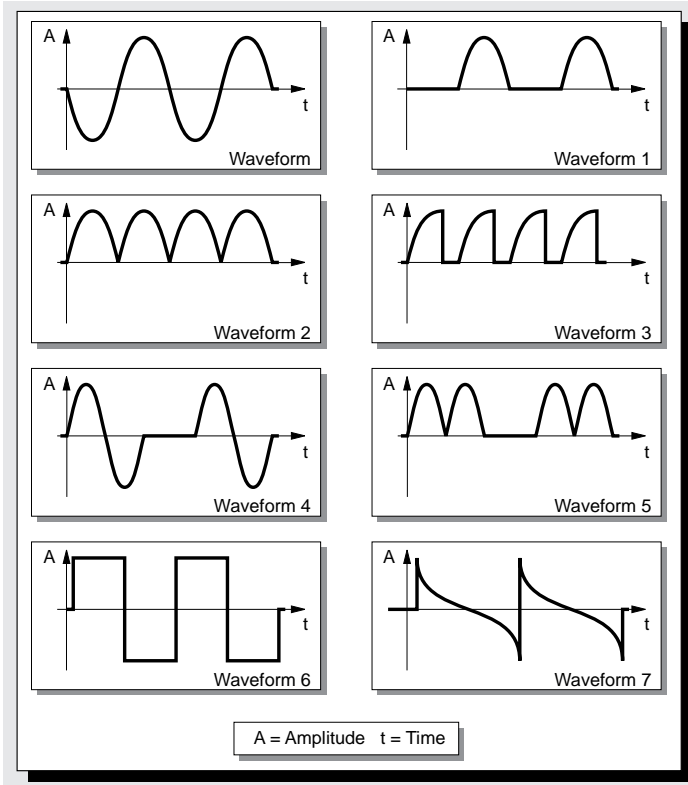
Amplitude2: Carrier
 Amplitude1: Modulator
 Frequency1: Modulator
 Frequency2: Carrier

This function says the carrier (*Amplitude2*) varies between $-Amplitude1$ and $+Amplitude1$ over time t . This means that $\sin(t+1)$ does not necessarily follow $\sin(t)$, but depending on the values of *Frequency* and *Amplitude* of the first Modulator, may be $\sin(t-5)$ or $\sin(t+100)$.

The illustration on the following page shows that you can generate very diverse signal patterns which are more similar to the natural signal progression of an instrument than are possible by simply adding two sine generators.

We've limited the discussion to coupling two sine generators. While it's possible to couple more generators to create more complex signal progressions, two is a practical limit. Here's why. The Yamaha OPL2 chip used in many sound cards has a two generator limit. A successor, the Yamaha OPL3, couples up to four sine generators, but the chip isn't found in many cards and isn't considered a Sound Blaster standard. So for the remainder of this chapter, we'll focus on the OPL2. Yamaha refers to the sine generators as operator cells.

Various Signal Progressions generated by frequency modulation



Making sound with operator cells and channels

An operator cell is the lowest level of sound generation. The OPL chip has eighteen such cells. Two cells - a carrier and a modulator - are used to generate one voice. Therefore nine different voices can be played simultaneously. A channel is the next higher level of sound generation. Two operator cells are paired to make one channel.

Besides the standard mode of nine voices, the OPL has a second mode of operation with six different voices (using twelve cells) and five percussion instruments (using the additional six cells). Which of the two modes you choose depends on your requirements. If you want to play a drum or other percussion instrument, you can use the second mode. On the other hand, the nine voice mode is preferable for playing classical music. We'll see how to select between modes shortly.

The Sound Blaster Pro sound card contains two OPL chips for a total of 36 cells for a total of 18 channels. To program the different chips, use a separate port address and output channel. While the first of the two chips is located at the original port address and becomes the left output channel of the Sound Blaster Pro card, the second chip is found at a different address and becomes the right output channel. Programming the two chips are identical except for the port address. You may want to limit yourself in programming only one of the OPL chips to be Sound Blaster compatible.

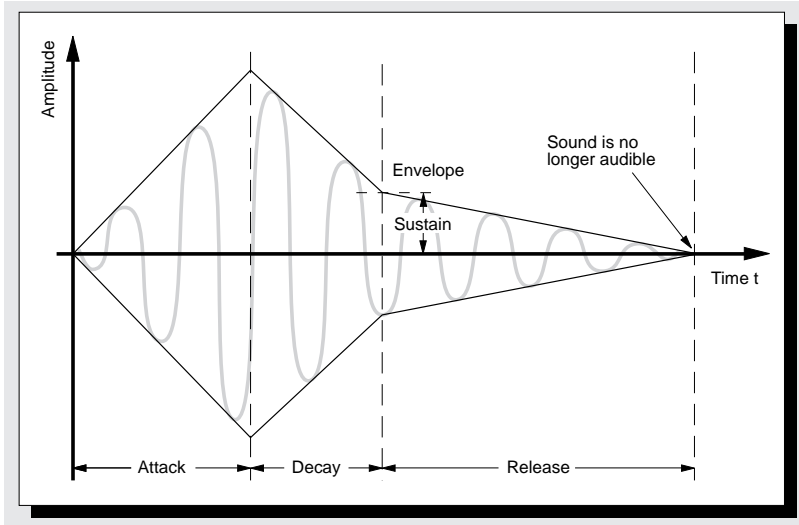
Experimenting with the different settings is the best way to understand the significance of the parameters. A program at the end of this chapter interactively tests and hears the results of various parameter settings.

The cell envelope

From the guitar string example, we know generating sound is more than just turning it on. A sound slowly builds until it reaches its greatest volume (amplitude). The volume remains at this level for a specific time depending on the instrument before fading. For example, the sound from a flute fades immediately but a sound from a bell continues for several seconds, although it rings more softly.

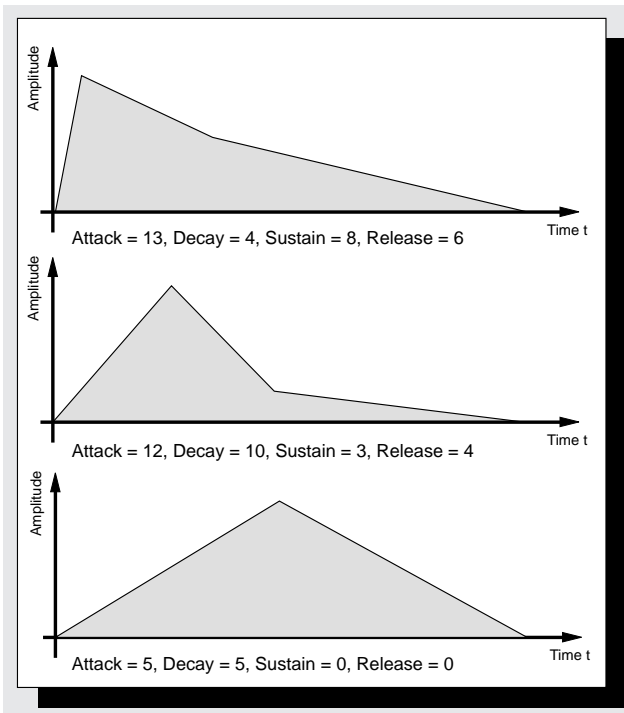
The *envelope* describes a voice's sound progression characteristics. Four parameters are used to specify the envelope for a voice: attack, decay, sustain and release. These parameters are collectively referred to as ADSR. For a sound card channel, the ADSR parameters can be set to values from 0 to 15.

Setting the envelope through Attack, Decay, Sustain and Release



The *attack* parameter is responsible for the first phase of sound generation. It specifies how quickly the sound reaches its full volume or amplitude. An attack parameter of zero represents a very gradual increase. An attack parameter of 15 means the sound immediately rises to maximum. The other parameters describe the other three phases of sound generation. These specify how quickly the sound fades away and falls from the maximum amplitude to a zero amplitude.

Different envelopes through different settings of attack, decay, sustain and release

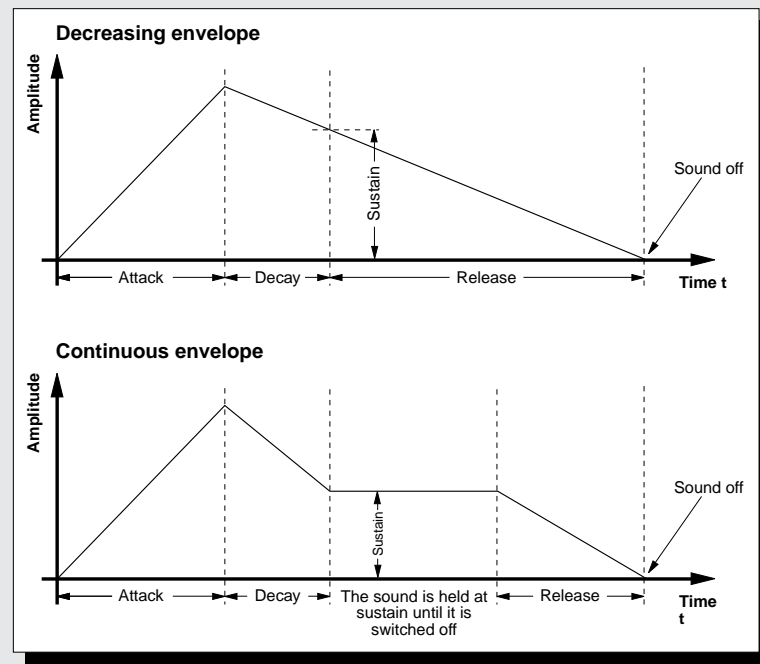


The *decay* parameter specifies the amount of time for the sound to fall from its maximum to the level specified by the *sustain* parameter. The greater the decay value, the faster the level falls to sustain level. A decay value of zero means the level falls quite slowly, while a value of 15 causes the level to drop almost immediately.

The *sustain* parameter specifies the amplitude level at which the transition between decay and release takes place. The sustain parameter refers to the relationship between maximum and zero amplitude. The greater the value, the shorter the decay phase.

The *release* parameter specifies the amount of time for the sound to fade. The greater the value, the faster the sound disappears. While a sound may no longer be heard at the end of the release phase (because the amplitude has reached the value 0), the generator cells nevertheless continues to operate. We'll discuss this again shortly.

Decreasing envelope versus continuous envelope



A second envelope type is the *continuous envelope*. A continuous envelope is also defined by the attack, decay, sustain and release parameters. However it differs from a decreasing envelope, described above, in one important way - the programmer can sustain the sound for any desired length. Like a decreasing envelope, you specify the duration of the attack and decay phases for the continuous envelope using the parameters for these variables. The OPL chip holds the sound at a constant amplitude in the sustain phase until the program switches off the sound through the appropriate channel. Only then does the sound move to the release phase and fade away.

The envelope you choose depends on the instrument you're simulating. Use the decreasing envelope with string instruments and all other instruments that fade by themselves. After all, there is no way to determine with this envelope the duration of the sound after making it. However, it's different with wind instruments, for example, which can hold their sounds until the musician is out of breath or the program ends. So, we recommend using the continuous envelope with this type of instrument.

When you use the decreasing envelope, you also have the option of setting automatic envelope reduction for high notes. This allows you to vary the sound for pianos and other stringed instruments, for example, where the high notes fade faster than the low notes.

The mute factor

The *mute factor* lets you lower the amplitude of an operator cell when you want one voice to be softer than the others. You can choose a setting between 0 and 63 according to following formula:

Mute = mute factor * 0.75 dB (where dB is Decibel measurement)

The multiplication factor

Use the *multiplication factor* to set the frequency ratio between the modulator and the carrier operator cells. While the frequency of the two cells can be different depending on the pitch of the desired tone, the ratio between modulator and carrier frequency for a given instrument is always the same. This is what gives an instrument its own characteristic timbre and is the reason the frequency is set for a channel, not for a generator cell. The frequency at which the cell oscillates is a result of multiplying the channel frequency by the multiplication factor of the cell.

The multiplication factor for a cell can be a value between 0 and 15. This value is multiplied by the frequency of the channel according to the table on the right. For example, the channel frequency with a setting of 0 would be cut in half.

To set an uneven ratio between the modulator and the carrier, e.g., 5:7, you must set a multiplication factor of 5 for the modulator and a multiplication factor of 7 for the carrier. However, the channel frequency compared with the actual desired frequency on the carrier must then be divided by 7. The OPL chip doesn't make the conversion, instead, the program must perform the computations.

The multiplication factor determines the frequency of a cell			
Multiplication factor	Frequency factor	Multiplication factor	Frequency factor
0	0.5	8	8
1	1	9	9
2	2	10	10
3	3	11	10
4	4	12	12
5	5	13	12
6	6	14	15
7	7	15	15

High mute

With the piano and other stringed instruments, the deeper the tone the louder the sound. The softer sounds appear closer to the end of the scale. Therefore, the OPL chip provides the option to mute notes in relation to their pitch. A setting of 0 to 3 according to the table on the right defines high mute.

Setting options for high mute			
Value	High mute	Value	High mute
0	None	2	1.5 dB per octave
1	3 dB per octave	3	6 dB per octave

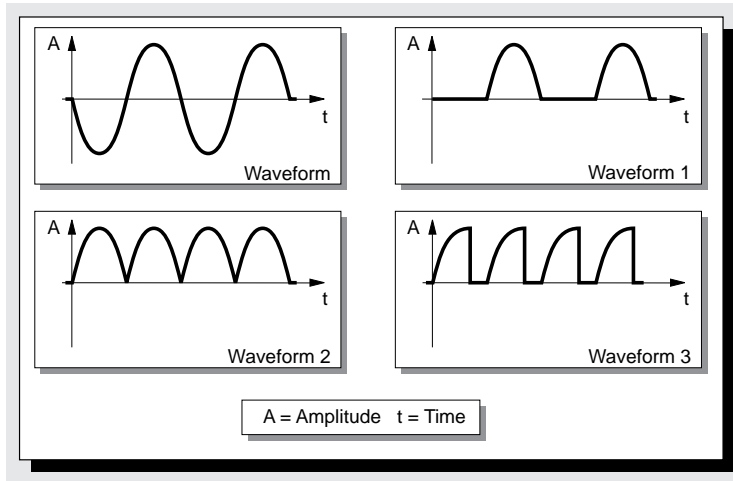
Tremolo and vibrato

The OPL has two additional features that are often found on home organs. These are tremolo and vibrato. These sound effects can be turned on or off for each operator cell. When tremolo is turned on, the sound volume rises and falls continuously at a frequency of 3.7 Hz. The volume level can be changed by either 1dB or 4.8 dB. When vibrato is turned on, the frequency of the generator cell is varied. For a cell frequency of 6.4 Hz, the frequency is increased by either 7% or 14% and then lowered. You can use vibrato to create a siren effect.

Waveform

A cell can be programmed to generate a waveform different from the normal sine wave. As Type 1, the generator cells continuously produce a zero level in the negative phase of the sine wave. As Type 2, the absolute value of the sine wave is continuously output. Type 3 is similar to Type 2, except the signal is shortened to zero in the falling phase of the wave. By varying the waveforms, it's possible to generate a completely different set of voices even though the other parameters remain constant.

Various waveforms as the output signal of the generator cells



Setting the generator cells

The OPL chip has about 120 different registers. These registers are used to set the generator cells, the channels and other characteristics for sound generation. Because the chip is a separate component of the sound card and the number of registers is rather large, not all registers appear in the I/O address range of the sound card. Instead, you communicate with the internal registers through the OPL's address and data register. These later two are accessible through the I/O ports.

- Ports 388h and 389h
To access the registers of the OPL chip, use ports 388h and 389h. These ports, first popularized by the AdLib card, are used by almost all sound cards that contain the OPL chip.
- First two ports of the Sound Blaster card, i.e., base address +0 and base address +1.
Accessing the OPL chip through 388h and 389h has become the preferred method for programming the sound card.

Accessing the OPL - address and data registers

To access any of the OPL's internal registers, follow this procedure. Write the number of the OPL register to the address port of the OPL chip: either port 388h or the SB-base address + 0. The OPL needs 3.3 microseconds to enable access to one of its internal registers. Therefore a machine language program may require a slight delay loop. One way to avoid this problem is to immediately read the port after writing to it. This insures the OPL has cleared the internal path to the desired register.

Now you can write to the internal register, by writing the desired data value to either port 389h or the SB-base address + 1. The OPL needs 23 microseconds to complete to write operation to the internal register. You'll have to consider in your programming tasks.

So writing to the internal registers is easy. However, you have no way to read the contents of the OPL registers. This limitation poses a few programming challenges especially for registers that are used to specify settings as a group of bits. For example, register 60h, is used to specify the attack and decay parameters of the envelope for generator cell 0. Since there is no way to read the contents of a register, how do you change one parameter without changing the other one?

One technique for handling this is to use "shadow RAM". With this method, the contents of the registers are held in an array. The OPL which has about 120 registers whose addresses are scattered over an address range from 0 to 255. By allocating 256 bytes, we can use call on a common routine to access the real registers. Let's call this routine SB_Write.

To keep things in sync, all accesses the OPL registers must be passed to the SB_Write function. This is because SB_Write not only writes the value to the desired register, but also copies the value to the shadow array. The shadow array now contains an accessible copy of the register contents.

To change single bits in a register, you simply have to read out the current contents from the shadow array, modify the bits in question and then use `SB_Write` to output the new contents to the OPL chip which then saves the contents in the shadow array. To ensure the shadow array actually represents the current contents of the various OPL registers at program start, you usually initialize all the registers and the entire array with the value zero. This suppresses sound generation and resets the OPL chip, since the defaults for all OPL settings are zero.

The sample programs at the end of this chapter give you a demonstration of how all of this works. However, right now let's get back to setting the operator cells.

The operator cell settings

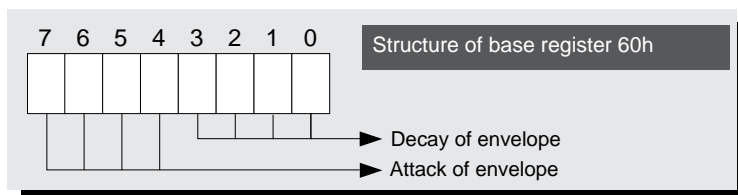
The following table lists the five base registers for setting the various parameters of the operator cells. Some of these registers hold several parameters. We'll describe their structure in later. The addresses specified in the table apply only to operator cell 0. For other cells, you must add an offset to each address, which results from the second table. By adding the offset, you get the number of the register used to set the desired parameter for a specific cell.

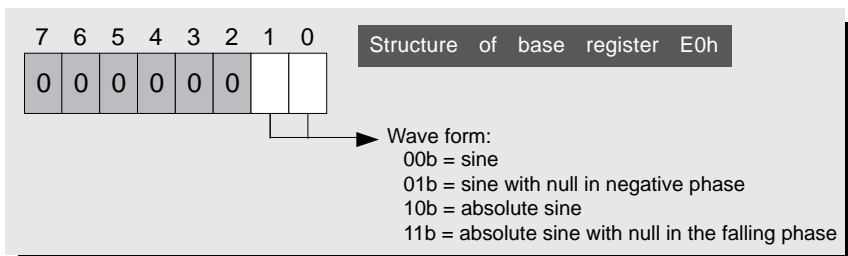
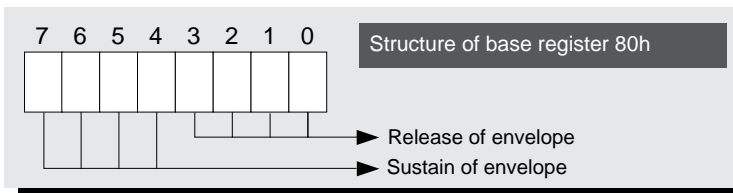
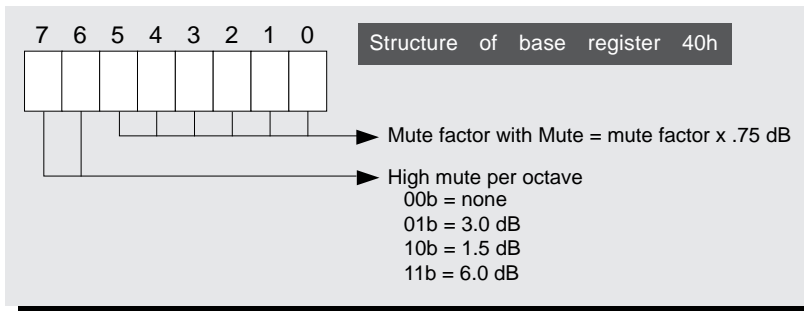
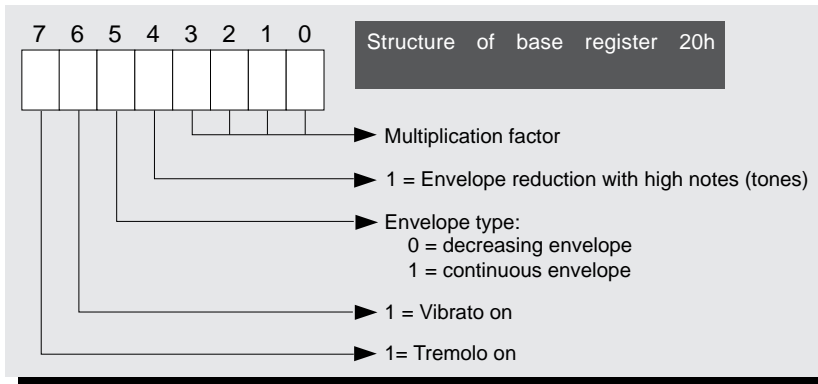
The base register for setting the cells	
Base register	Contents
20h	Multiplication factor, Envelope reduction and type, Vibrato, Tremolo
40h	Mute factor, High mute
60h	Envelope: Attack, Decay
80h	Envelope: Sustain, Release
E0h	Waveform

Offsets for the different registers used to set the generator cells					
Operator cell	Offset	Operator cell	Offset	Operator cell	Offset
0	00h	1	01h	2	02h
3	03h	4	04h	5	05h
6	08h	7	09H	8	0Ah
9	0Bh	10	0Ch	11	0Dh
12	10h	13	11h	14	12h
15	13h	16	14h	17	15h

Notice the offset for operator cell 10 is 0Ch. To set the attack and decay envelope parameters for this cell, add offset 0Ch to the base address 60h of this register. This yields register 6Ch, in which the desired setting is written. To determine the offset for a cell, create a static array within the program in which the offsets of the individual operator cells are stored. To access one of these cells, use the cell number as the index into the array, which gives you the required offset address. Then add the offset address to the base address for the cell.

The following five illustrations show the interrelationship between registers and operator cells:





Setting the channels

We said that two operator cells are paired to make a single channel. Therefore, the OPL's 18 generator cells allow for up to nine channels. This means that up to nine different voices can be programmed independent of one another.

The following table shows how the OPL channels are organized. The relationship between a channel and its two operator cells is predetermined. To program the channels, create a static array in which you store the numbers of the carrier and modulator operator cell. Then use the channel number as an index into the array to determine the carrier and modulator cell numbers.

Organization of the operator cells and channels					
Channel	Operator cell - Modulator	Operator cell - Carrier	Channel	Operator cell - Modulator	Operator cell - Carrier
0	0	3	1	1	4
2	2	5	3	6	9
4	7	10	5	8	11
6	12	15	7	13	16
8	14	17			

The Channel parameters

Five different parameters can be set for each channel. These parameters define the tone to be generated and the mode of operation for the operator cells. Block number and frequency are the first two parameters. Block number is the octave from which a note is to be played. The OPL chip supports eight octaves, numbered from 0 to 7. A value of 0 represents the lowest octave, while 7 represents the highest octave. To switch from one octave to the next, simply increase the value in the corresponding register.

Frequency is set in relation to the current block number or octave. Although an octave consists of only twelve harmonic tones, the frequency can be set to values between 0 and 1023. This means the resolution of the OPL chip far exceeds that of a normal music keyboard. The following table shows corresponding values of frequency and harmonic tones. Other frequency values are permissible if you want to experiment.

Settings for the frequency parameter of a channel								
Tone	Value for frequency parameter	Freq (Hz)	Tone	Value for frequency parameter	Freq (Hz)	Tone	Value for frequency parameter	Freq (Hz)
C#	363	277.2	F	458	349.2	A	577	440.0
D	385	293.7	F#	485	370.0	B	611	466.2
D#	408	311.1	G	514	392.0	H	647	493.9
E	432	329.6	G#	544	415.3	C	686	523.3

Another parameter specifies the type of connection between the two operator cells of a channel. One type of connection is the coupling of two cells as described earlier which is the basis of FM synthesis. Another type of connection is to combine the output of two cells through addition. This generates two overlapping oscillations which can generate some interesting voices and effects.

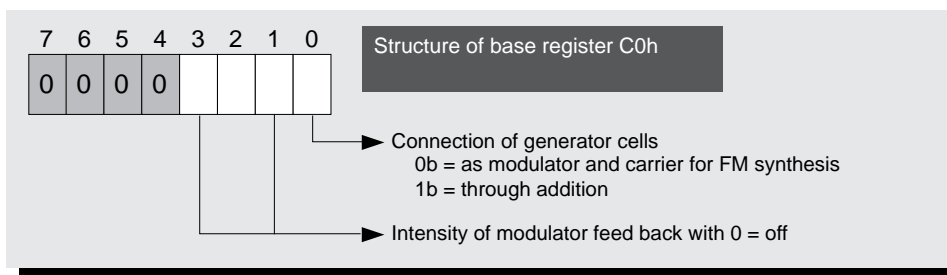
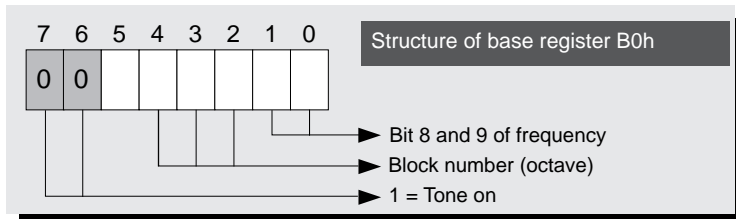
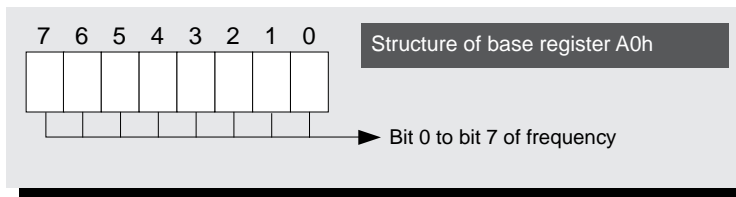
Still another parameter tells the modulator operator cell to add a portion of its output signal back to the input. A value between 0 and 7 determines the intensity of the feedback. The default value of zero disables feedback. Because the effect of this feedback is quite difficult to describe, we simply recommend that you experiment with this parameter. A good starting point is to use the sample program at the end of this section.

A final parameter tells the channel to either start or stop an operation. The next section shows you how to set the various parameters using the registers of the OPL chip.

Setting the channel parameters

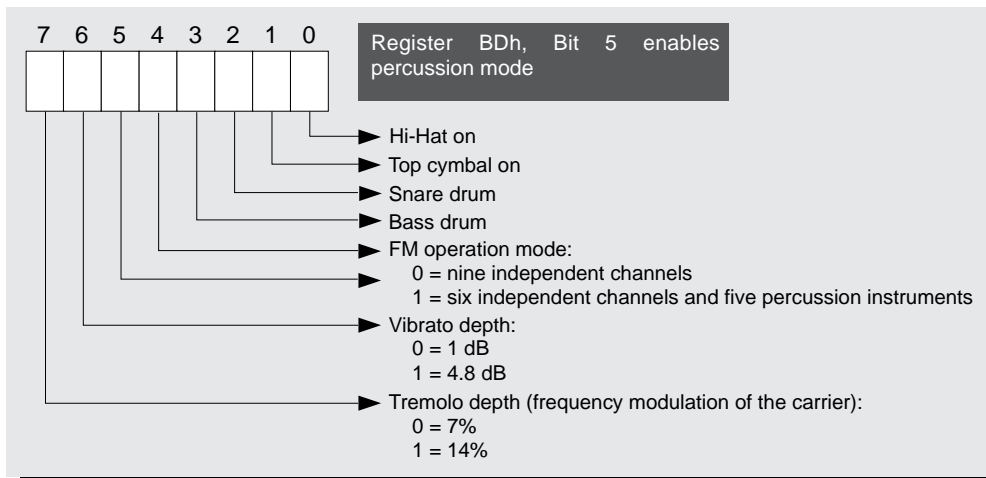
Like the operator cells, the channel parameters are also set using different base registers, to which an offset must be added for the desired channel. Unlike operator cells, you don't have to look up the offset in a table, because the channel number itself is the offset. To program channel 7, simply add 7 to the base address of the appropriate channel registers.

The five different channel parameters are contained in three base registers with addresses A0h, B0h and C0h. The frequency parameter is contained in two registers, since ten bits are required to represent its range (0-1023).



Percussion mode

In addition to the default mode with nine independent voices (channels), the OPL chip has a second mode in which only six independent voices are available. However, five percussion instruments are also available. Most games with sound accompaniment operate in this mode because the percussion instruments provide the necessary effects. Therefore, the limited number of voices isn't a problem. Register 0BDh, bit 5 switches between percussion and nine voice modes.



The above illustration also shows that register BDh also specifies the settings for the tremolo and vibrato depth. These values affect all generator cells that have tremolo or vibrato enabled. For now, let's return to percussion mode.

The table on the right shows the interrelationship between the percussion mode instruments and cells.

The channel and operator cell assignments have not changed compared to standard mode. One slight difference is that, except for the bass drum, two operator cells share a single channel. To enable sound generation of the percussion instruments you must set the appropriate bit in register 0BDh, rather than enabling the channel. In fact, the on/off bit in the channel register for the percussion instruments must also be set to zero; otherwise, the percussion instruments will remain silent.

Percussion Mode instrument, operator cells and channels		
Instrument	Operator cell	Channel
Bass Drum	12 + 15	6
Hi Hat	13	7
Tom Tom	14	8
Snare Drum	16	7
Top Cymbal	7	8

Since two percussion instruments share a channel, only one block number and one frequency register are available for both instruments. As a result, the two percussion instruments of a channel will play in the same pitch, provided they are played at the same time. The multiplication factor of the operator cell allows for varied tuning.

FM Programming

All this information about the various registers provides only a theoretical description of how the OPL operates. To really learn how it works, you'll have to experiment and see how the different register contents affect the sound generation. The FM program, at the end of this section, lets you interactively set many of the registers and use a keyboard to elicit sound from the card. If you take the time to experiment with FM, you'll quickly develop a feeling for the numerous ways that you can influence the sound card. FM works with Sound Blaster and other compatible cards containing an OPL chip at ports 388h and 389h.

FM itself includes only the code for interacting with the user. Two other modules, SBUTIL and FMUTIL, control the sound card. Any program can use the include files for these modules (SBUTIL.C or FMUTIL.C) to use the functions contained in them. The SBUTIL module is used often in this chapter since it contains some very basic routines for programming Sound Blaster cards. FMUTIL is intended specially for programming the OPL chip.

Functions in module SBUTIL	
Function	Task
sb_GetEnviron	Initialize SB_base structure using the SB environment variables
sb_Print	Display SB_base structure
sb_LoadDriver	Load Sound Blaster driver into memory
sb_UnloadDriver	Remove Sound Blaster driver from memory

SB_base structure listed in the table for the sb_getEnviron and sb_Print functions refers to the data structure of the same name in the SBUTIL.H include file. Calling sb_GetEnviron, initializes the structure with the important parameters about the installed Sound Blaster card.

All the functions in SBUTIL and FMUTIL that directly access a register of the Sound Blaster card reference the port address of the sound card from this data structure.

Therefore, you must call sb_GetEnviron before the other two modules. FMUTIL is given this information by fm_SetBase.

You'll find the following program(s) on the companion CD-ROM



SBUTIL.H (C listing)
 SBUTIL.C (C listing)
 FMUTIL.H (C listing)
 FMUTIL.C (C listing)
 FM.C (C listing)
 FM.PAS (Pascal listing)
 FMUTIL.PAS (Pascal listing)
 SBUTIL.PAS (Pascal listing)
 DSPUTIL.PAS (Pascal listing)
 ARGS.PAS (Pascal listing)
 WIN.PAS (Pascal listing)

FMUTIL contains functions for controlling all aspects of FM synthesis. FMUTIL uses a shadow array (Mirror array variable) to keep track of the current OPL register contents (see the following table). However, remember the reliability of this array is assured only if you use the FMUTIL functions `fm_Write` and `fm_WriteBit` to change the various OPL registers.

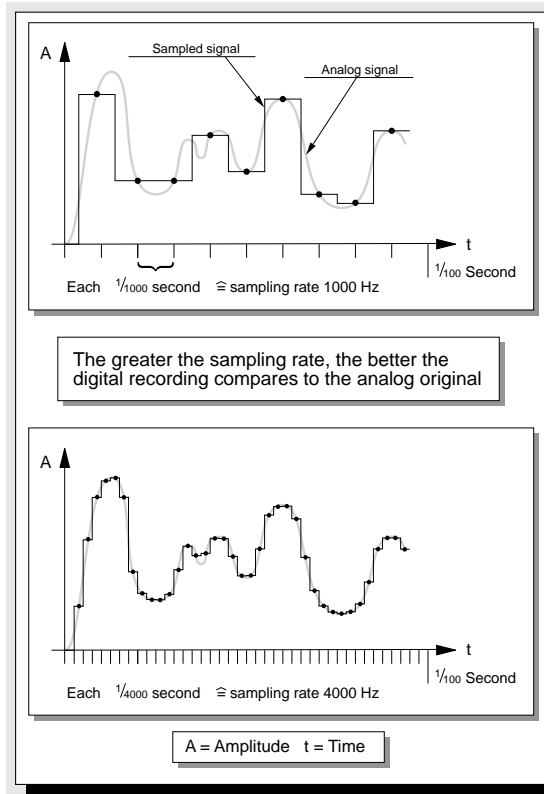
Functions in module FMUTIL	
Function	Task
<code>fm_SetBase</code>	Provide routines with an initialized SB_base structure
<code>fm_Write</code>	Write value in an OPL register
<code>fm_WriteBit</code>	Set or Clear a bit in OPL register
<code>Reset_Card</code>	Reset all OPL registers
<code>fm_GetChannel</code>	Get channel number of an oscillator
<code>fm_GetModulator</code>	Get modulator number of a channel
<code>fm_GetCarrier</code>	Get oscillator number of a channel
<code>fm_SetOscillator</code>	Sets all the parameters of an operator cell
<code>fm_SetModulator</code>	Sets all the parameters of the modulator
<code>fm_SetCarrier</code>	Sets all the parameters of a carrier
<code>fm_SetChannel</code>	Sets all the parameters of a channel
<code>fm_PlayChannel</code>	Switch channel on or off
<code>fm_SetCard</code>	Set tremolo and vibrato depth for all channels
<code>fm_PollTime</code>	Time delay
<code>fm_PlayHiHat</code>	Switches hi hat on or off
<code>fm_PlayTopCymb</code>	Switches top cymbal on or off
<code>fm_PlayTomTom</code>	Switches tom tom on or off
<code>fm_PlaySnareDru</code>	Switches snare drum on or off
<code>fm_PlayBassDru</code>	Switches bass drum on or off
<code>fm_PercussionMo</code>	Switches between nine channel mode and percussion mode

Sampling

Sampling is a method of recording analog sounds by converting them into a series of digital pulses. You can record these analog sounds through a microphone or from stereo equipment for example, and save the sequences of pulses on your hard drive. At a later time you can replay the recording without a decrease in audio quality. Sampling is the same technique that is used to record audio CDs in a sound studio.

Basically, sampling is a high quality digital to analog or analog to digital conversion. During recording, the sound card receives the analog sound signal and converts it into digital "samples". Recording is therefore an analog to digital conversion. When you play back the sample, the digital data stream of the sample is converted back into an analog signal.

How well the results of this analog to digital conversion match the original analog signal depends on the sample frequency and its resolution. Ideally, the analog signal is sampled infinitely. Of course this is impossible and would require an infinite amount of disk space. Instead, a specified number of samples are taken per second. At continuous intervals the current analog signal is converted to a digital value. The more frequently this happens and the shorter the intervals between single samples, the higher the recording quality. For audio CDs, the sampling rate is 44.1 KHz. This means that 44,100 samples per second are taken.

Samples of signals

Besides the sample frequency, the resolution of the sample determines the quality of digital recordings. Resolution is the range of values used to represent the analog sound signal on a digital scale. For audio CD's, sound is recorded at a level of 16-bits. So it's possible to differentiate between 65,536 signal intensities. If a lower resolution is used (for example, 8-bit resolution) then the width is 256 times smaller, a value which your ears can clearly detect.

Sampling lets you record any natural sounds. You can edit the sounds with appropriate software tools. You can then send the sounds to other programs such as games. You can record many more sounds and much more realistic ones than are possible using a synthesizer such as the OPL. This is an overriding advantage of sampling compared to synthetic sound production.

One of the drawbacks of sampling is that digitized sound requires a large amount of disk space. For example, one second of CD quality music requires about 170K; ten seconds requires 1.7 Meg and a minute requires about 10 Meg of disk space. A complete CD with 72 minutes of music needs 600 Meg

With these large storage requirements, the CD-ROM is the only practical way to distribute software that uses large amounts of sampled sound.. But even with CDs, it's still necessary to find ways to reduce the volume of data, for example, by reducing the sampling frequency. Doing this is usually acceptable because many sounds are still realistic even when recorded at much lower sampling rates. The rule-of-thumb minimum is 12 KHz; at this level, the human ear can clearly distinguish the loss of sound quality.

Next we'll discuss how to record and playback sound using the Digital Sound Processor (DSP) on the Sound Blaster card.

DSP Basics

All Sound Blaster cards have a built-in digital sound processor. First, some background information.

The various DSP versions

With each new generation of Sound Blaster card, its digital sound processor has become more powerful. In fact, you can identify the different Sound Blaster cards by knowing which version of DSP it uses. Later, we'll see how you can determine which version of DSP is on a card from a program.

The following table lists the cards and DSP versions in the Sound Blaster family:

SB Card	DSP version
SB 1.5	1.xx - 2.00
SBMCV (Sound Blaster for the MicroChannel)	1.xx - 2.00
SB 2.0	2.01+
SBPRO	3.xx
SBPRO MCV (Sound Blaster Pro for the MicroChannel)	3.xx
SB 16	4.xx
SB 16 with ASP (Advanced Signal Processor)	4.xx

The DSP first appeared with the Sound Blaster and Sound Blaster 1.5 (both are now called Sound Blaster 1.5). These cards have various versions of the DSP versions including Version 1.xx and Version 2.0. From a programmer's viewpoint, there are no differences among the various 1.xx versions. This is also true with Versions 3.xx and 4.xx. DSP Version 2.01, used in the Sound Blaster 2.0 card, features a "High Speed mode". Version 3.xx also has this "High Speed mode". However, this feature is missing from DSP Version 4. The Sound Blaster Pro card uses DSP Version 3. The SB 16 and its upgrades use the current version, DSP Version 4.xx.

The main differences among the various versions of the DSP chip is sampling speed and bit depth. Speed refers to the number of samples that can be recorded or played back per second. Bit depth refers to the resolution of the sample - the range of values used to represent the original analog signal on a digital scale. A higher resolution yields a higher quality reproduction. In abbreviated terms, the quality of an audio CD is "16-bit, stereo, 44.1 KHz". This means the original is recorded by sampling 44,100 times per second, using 16-bit values to represent the original analog signal intensity; two channels are recorded simultaneously - the left and right channels. This amounts to a data rate of 170K per second (44,100 samples per second * 2 bytes per sample * 2 channels).

The following tables compare the quality levels for the different DSPs:

Sampling rates of different DSPs for output				
DSP Version	Mono/Stereo	High-Speed	Data Format	Sampling Rate
All DSPs	Mono Mono Mono		8-bit/4-bit ADPCM* 8-bit/3-bit ADPCM 8-bit/2-bit ADPCM	4000-12000 Hz 4000-13000 Hz 4000-11000 Hz
1.xx and 2.00	Mono		8-bit	4000-23000 Hz
2.01+	Mono Mono	Yes	8-bit 8-bit	4000-23000 Hz 23000-44100 Hz
3.xx	Mono Mono Stereo	Yes Yes	8-bit 8-bit 8-bit	4000-23000 Hz 23000-44100 Hz 11025-22050 Hz
4.xx	Mono Mono Stereo Stereo		8-bit 16-bit 8-bit 16-bit	5000-44100 Hz 5000-44100 Hz 5000-44100 Hz 5000-44100 Hz

Sampling rates of different DSPs during recording				
DSP Version	Mono/Stereo	High-Speed	Data Format	Sampling Rate
1.xx and 2.00	Mono		8-bit	4000-13000 Hz
2.01+	Mono	Yes	8-bit	4000-13000 Hz
	Mono		8-bit	13000-15000 Hz
3.xx	Mono	Yes	8-bit	4000-23000 Hz
	Mono		8-bit	23000-44100 Hz
	Stereo		8-bit	11025-22050 Hz
4.xx	Mono		8-bit	5000-44100 Hz
	Mono		16-bit	5000-44100 Hz
	Stereo		8-bit	5000-44100 Hz
	Stereo		16-bit	5000-44100 Hz

High speed mode in DSP versions 2.01+ and 3.xx uses a higher sampling rate than earlier versions. But as mentioned, this feature was removed in DSP Version 4 which attains CD quality reproduction without this mode.

DSP transfer modes

The DSP chip supports five different modes of operation. In one mode, the DSP operates using a polling technique. For the other four modes, the DSP operates in conjunction with the DMA controller. The advantage of the four DMA modes is the data can be transferred in the background while the CPU performs other tasks. When the data transfer is complete, the DSP signals the CPU using an interrupt. The following summarizes the five different modes:

Direct mode

In direct mode, the DSP operates by polling. Data is passed to or from the DSP, one byte at a time, by a program. The sampling rate is determined by the speed at which the program runs. The samples are limited to eight-bit unsigned values. To achieve a constant sampling rate, the data is passed to or from the DSP using an interrupt routine that is tied to the timer.

An advantage of direct mode is its ease of programming. Since you don't have to interface with the DMA controller, writing code for direct mode is quick and dirty. A disadvantage of direct mode is that it cannot achieve anywhere near the sampling rates possible using the other DMA modes. In fact, the sampling rate is limited to about 15,000 Hz which limits the quality.

Single cycle DMA mode

In single cycle mode, the DMA controller handles the data transfer between the DSP and RAM. Both the DMA controller and the DSP are programmed separately for each sample block transferred. Based on the type of DMA controller, a block can be a maximum of 64K or 128K. The entire data block must fit within a single 64K memory page (or within one 128K memory page while using a 16-bit DMA channel). After the transfer is complete, the DSP triggers an interrupt. As its name suggest, this mode is used to transfer a single sample block between RAM and DSP. As with the other DMA modes, the DSP can process 8-bit samples, 16-bit samples and ADPCM samples.

Auto initialize DMA mode

Auto initialize mode can handle the high data rates needed for CD-audio quality - 16-bit stereo samples at 44.1 KHz.. This mode takes advantage of the DMA controller's ability to initialize itself with the original starting address and block length after a block transfer (Auto-Init-Mode). This makes it possible to edit samples of almost any length. The technique used to manage Auto initialize DMA mode is commonly called "double buffering". Once the DSP is in auto initialize DMA mode, there are two ways to halt the data transfer:

- By sending the DSP a command for executing a single cycle DMA mode transfer. The DSP completes the current block transfer and then perform the single cycle DMA mode transfer.
- By sending the DSP a command to exit auto initialize mode. Here too, the DSP completes the current block transfer and then exits auto initialize DMA mode.

High speed DMA mode

To reach higher sampling rates than can be achieved with auto initialize DMA mode, you can use high speed DMA mode found in DSP Versions 2.01+ and 3.xx. High speed DMA mode can perform both single cycle DMA transfers and auto initialize DMA transfers (for 8-bit mono and stereo samples only).

The one drawback of high speed DMA mode is that this mode can only be terminated by reset after the transfer is complete. Because this mode is supported only by DSP versions 2.01+ and 3.XX, it's not very commonly used. Beginning with version 4, this mode isn't needed since the new DSP chip is capable sampling at a 44.1 KHz rate in auto initialize DMA mode.

ADPCM DMA mode

Earlier we touched on the tremendous volumes of data produced by sampling. To reduce the amount of data you can compress it. One compression technique called ADPCM (Adaptive Delta Pulse Code Modulation), records the difference between samples rather than the value of each sample. Sound Blaster cards using DSP Version 1 can playback samples that have been compressed using the ADPCM.

The Sound Blaster card recognizes these ADPCM formats: 8/4, 8/3 and 8/2. Format 8/4 uses four bits to specify the difference from the previous sample. Therefore two samples can be packed into a single byte which reduces the space requirements of the sample block in half. The limitation is the change compared to the previous sample can only range between -7 and +7, since four bits are available and one is used for the sign, leaving three bits for the difference. Sample data containing larger changes cannot be compressed perfectly with ADPCM 8/4.

Different DSP operating modes of the Sound Blaster family					
Mode	DSP Version				
	1.xx	2.00	2.01+	3.xx	4.xx
8-bit mono single cycle	x				
8-bit mono auto initialize		x			
8-bit mono ADPCM single cycle	x				
8-bit mono ADPCM auto initialize		x			
8-bit mono high speed single cycle					
8-bit mono high speed auto initialize					
8-bit stereo high speed single cycle					
8-bit stereo high speed auto initialize					
8-bit/16 bit mono single cycle					
8-bit/16 bit mono auto initialize					x
8-bit/16 bit stereo single cycle					x
8-bit/16 bit stereo auto initialize					x

For higher compression, you can use ADPCM formats 8/3 and 8/2. With these, however, the differences between the sampling values must be even lower, because only 3 (8/3 mode) or 2 bits (8/2 mode) remain to record the difference between samples.

The three ADPCM formats are used only for 8-bit samples. Unless you're using a SB 16 card with ASP, you cannot directly record using the ADPCM formats. Creative Labs hasn't released the exact specifications for ADPCM samples, so there is no easy way to convert raw, uncompressed recordings into ADPCM formatted recordings.

Transfer modes supported by the different DSPs

The table above lists the transfer modes which are available to the different versions of DSP. Notice that CD quality recording with 16-bit samples is possible with DSP Version 4.xx (found in Sound Blaster 16).

DSP and DMA controller interaction

In the different DMA transfer modes, the DMA controller and the DSP work hand in hand. For example, when recording, the DSP receives the analog signals from a sound source, converts them into digital samples, and then transfers them to a buffer in RAM with the help of the DMA controller. Because of the limitations of the DMA controller, the transfer buffer must be located in the first Meg of RAM and cannot exceed a 64K page limit. Conversely, when playing samples, the DMA controller transfers the data to the DSP from a buffer in RAM. The DSP converts the sample data into analog signals and passes the signals to the Sound Blaster card amplifier. From there, the signals are heard as sound through a set of headphones or speakers.

The interaction between the DMA controller and the DSP is controlled by the sampling software. The software must first program the DMA controller before recording or playing back the sample. The settings for the Sound Blaster card's DMA channel are found in the BLASTER environment variable. The D parameter is the channel number for 8-bit transfer; the H parameter is the channel number for 16-bit transfer. Remember, 16-bit transfer is a feature of the Sound Blaster 16 with a DSP Version 4.xx. Next the software specifies the starting address of the transfer buffer and the block length for the DMA controller. To perform auto initialize DMA mode transfers, you must also switch the DMA controller channel to auto init mode. The DMA controller is ready to go.

Now on to the DSP side. The DSP needs several items of information. One is the sampling frequency; a second is the length of the data to be transferred from memory to the DSP or from the DSP to memory; a third is the source of the sound samples or the destination for the sound output. Now the software sends a special command to the DSP to start the recording or playback of the samples.

After processing the last sample, the DSP triggers an interrupt informing the sampling software the transfer is complete. The interrupt is again found in the BLASTER environment variable as parameter I. As a rule, the sampling software provides its own interrupt handler for the interrupt, which can recognize this event and then continue working. We'll discuss the details of programming the DSP later.

Double buffering

Double buffering is a software technique which "converts a PC into a DAT drive". It's a technique which lets you capture or play back samples of nearly any length. You can use double buffering (sometimes called ping-pong buffering) to record samples from any sound source onto the hard drive or to play back prerecorded samples previously recorded on the hard drive. To use this technique, first allocate a DMA buffer. Recall that a DMA buffer has a maximum size of 64K and is located within a 64K memory page.

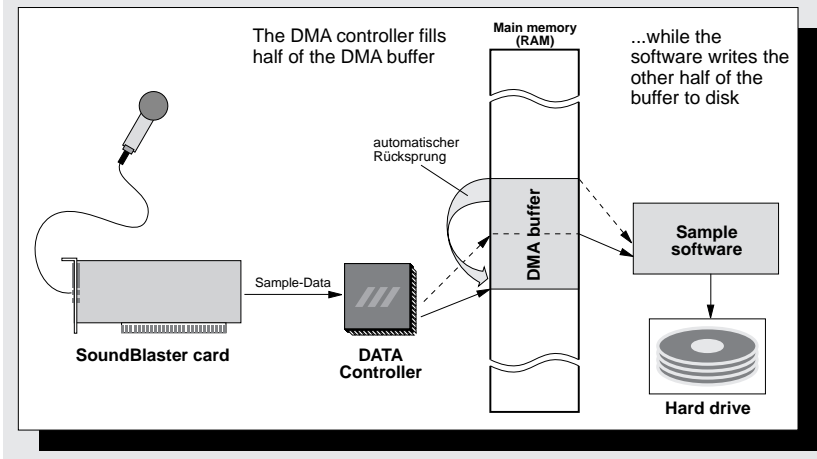
During recording, the samples are transferred from the DSP to this buffer. During playback, the samples are transferred from this buffer to the DSP. Prior to either recording or playback, the DMA controller is programmed with the starting address, the length of the block and to operate in auto init mode. Each time the buffer is filled (during capture) or emptied (during playback), the operation is repeated. Once started, the DMA controller repeats the operation over and over.

With double buffering, the DSP is programmed to transfer only half of the length of the block. During recording, the DSP fills only one-half of the buffer with sampled data and then triggers an interrupt. The interrupt handler sets an internal flag and then commands the DSP to write additional sampled data to the second half of the buffer and completes its task.

During playback, the DSP reads and plays the data from the first half of the buffer and then triggers an interrupt. The interrupt handler sets an internal flag and commands the DSP to read additional data from the buffer and is finished. Since the DMA controller was not reprogrammed, only half of the transfer has been complete. Therefore the controller continues from the point where it left off - at the first byte of the second half of the buffer.

While the DSP reads from or writes to the second half-buffer in the background, critical work is taking place in the main program. Since the start of sampling, the program has been looping continuously. This purpose of the loop is to check the flag set by the interrupt handler. If set, then the DSP has jumped from the first half-buffer second half-buffer. For recording, the program writes the completed half-buffer to the sample file. For playback, the program loads the next half-buffer with the next block of data from the sample file.

*Sampling in
Auto-Initialize-
DMA-Mode with
double buffering*

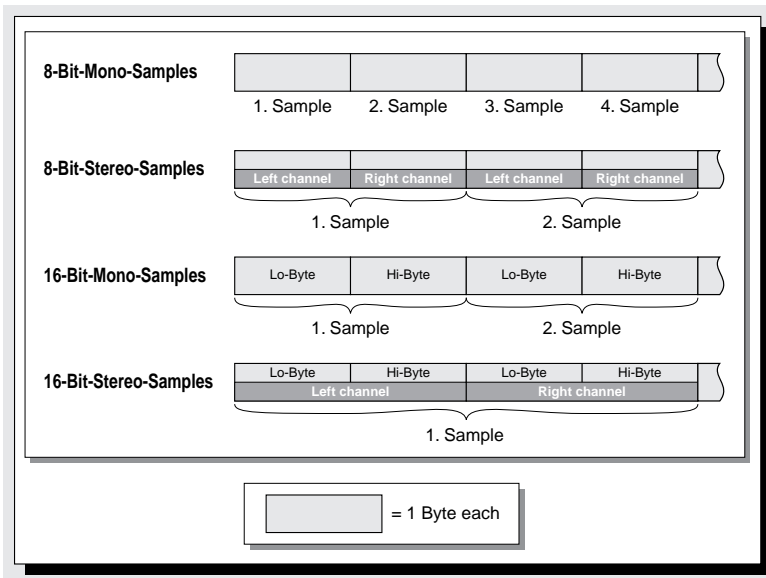


When the DSP completes the processing for the second half-buffer, the DMA controller jumps back to the beginning of the buffer to repeat the next transfer from the first half-buffer again. This ensures the program can always reload or save the half-buffer which the DSP is not processing and the DSP can move from one half-buffer to the other half-buffer without being disturbed. So, the DSP and DMA controller access the first and second half-buffers alternately. This method of course assumes the main program runs fast enough to reload or save the idle half-buffer before the DMA controller returns again to the beginning of this same half-buffer. This is no problem for a 486 system if the modules and subroutines are programmed intelligently. The program at the end of this section and on the companion CD-ROM uses this technique.

Sample data structure

The DSP writes the sample data to RAM in a specific format. This same format is used later to play back the samples. The exact sequence of the samples depends on whether the samples are recorded in mono or stereo and whether the samples are 8-bit or 16-bit. The DSP can handle both unsigned and signed values. 8-bit samples are unsigned while 16-bit samples can be either signed or unsigned. An unsigned sample is recorded as a positive number. An 8-bit unsigned value ranges from 0 to 255 (00h to ffh); a 16-bit unsigned value ranges from 0 to 65,535. A 16-bit signed value ranges from -32,768 to +32,767 (-8000h to 7FFFh).

*Structure of
sample data in
memory*



Accessing the DSP

The following table lists the five different registers that control the DSP. Their addresses are relative to the base address of the Sound Blaster card.

DSP ports				
Port	Name	Read	Write	Purpose
+06h	Reset		x	Resets the DSP.
+0Ah	Read Data	x		Reads data from the DSP.
+0Ch	Write Command / Data		x	Writes commands and data to DSP.
+0Ch	Write Buffer status	x		Indicates whether DSP is ready to receive commands or data.
+0Eh	Read Buffer status			Indicates whether data can be read from the DSP.

Sending data and commands to the DSP

The assembly language OUT instruction sends commands and their parameters to the DSP. However, you cannot write to the port unless the Write-Buffer status port is empty. If the DSP is still processing a previous command, the status port contains the last command or data value. Reading the Write-Buffer status port to determine if bit 7 is set. If so, the DSP is still busy. You can then loop until the Write-Buffer status port is clear. Only then should you send the next command or data to the DSP.

The following is the assembly language code:

```

        mov     dx,SbPortBase      ;Load port address of SB card
        add     dl,0Ch             ;Offset for Write Buffer Status
Wait:   in      al,dx              ;Read the Write Buffer Status Reg
        test    al,80h             ;Check bit 7
        jne     Wait              ; If not 0, then one more pass
;--- DSP is now ready to receive the next instruction
        mov     al,CommandByte     ;Load DSP command
        out     dx,al              ;and output

```

Several DSP commands require parameters. Both the command code and the parameters are passed to the DSP through the port. The command code is always sent first. Then the parameters are transferred to the DSP one at a time, in a predetermined order depending on the command. Here again, you must wait for the DSP to tell you its ready to accept the next byte using the Write-Buffer status port.

Reading data from the DSP

Reading data is similar to the writing to the DSP. In this case, the Read-Buffer status register is used to signal when the requested data is ready. The difference is that ready is indicated when bit 7 of this register is 1, not zero. The following is the assembly language code:

```

        mov     dx,SbPortBase      ;Load port address of SB card
        add     dl,0Eh             ;Offset for Read Buffer Status
Wait:   in      al,dx              ;Read the Read Buffer Status Reg
        test    al,80h             ;Check bit 7
        je      Wait              ;If not 1, then one more pass((blank line paragraph))
;--- DSP is now ready to accept the next command
        sub     dl,4               ;from Read Buf Status Reg. to
                                   ;Read Data Register
        in      al,dx              ;Read the Read Data Register

```

DSP reset

A program should always begin by resetting the DSP. This resets all the internal registers and also returns information as to whether a Sound Blaster card and a DSP are actually located at the specified port address.

Here are the steps for doing this:

1. Write the value 1 to the Reset port.
2. Wait at least three microseconds (the DSP requires this long to respond).
3. Write the value 0 to the Reset port.
4. The byte sequence 0 and 1, tells the DSP to perform a reset. When complete, and the Read-Buffer status register goes to 1, the DSP writes the value 0AAh to the Read-Buffer status port. Since you still don't know at this time whether a DSP is actually present, you should include a time out loop; otherwise you may be waiting a very long time.

The following is the assembly language code for reset:

```

;--- first send 1 to reset port -----
        mov     dx,SbPortBase    ;load port address of SB card
        add     dl,6             ;Offset for reset port
        mov     al,1
        out     dx,al    ;--- give DSP so time time
                                ;--- Loop 256 times
        sub     al,al            ;Set AL to 0
DspLoop1: dec     al             ;Decrement AL
        jne     DspLoop1        ;not 0, then loop again

        out     dx,al            ;send second byte (0) to DSP

;--- wait for Read Buffer Status Register to -
;--- indicate availability of a byte
        xor     cx,cx            ;maximum 65536 loops
        add     dl,8             ;from reset port to Read Buffer status register
DspLoop2: in      al,dx           ;read Read Buffer status register
        test    al,80h           ;test bit 7
        jne     ByteAvail        ;if set, then byte available
        loop    DspLoop2         ;if not, then next loop run
        jmp     NoDSP            ;if there's still no byte after 65536 runs

ByteAvail: ;--- a byte is available at Read data port --
        sub     dl,4             ;from Read Buffer Status to Read Data
        in      al,dx           ;read out Read data register
        cmp     al,0AAh          ;the DSP must answer with 0aah
        jne     NoDSP            ;if not, then no DSP present

;--- if the processor comes here, a DSP is present
        jmp     DSPFound
NoDSP:    ;--- could not find DSP, continue error processing

```


Handling interrupts

The Read Buffer status register plays an important part in handling DSP interrupts. If the DSP triggers an interrupt following the processing of a sample block, the interrupt handler must indicate to the DSP that it received the interrupt. Before DSP Version 4.xx, reading the Read Buffer status register was sufficient for this purpose.

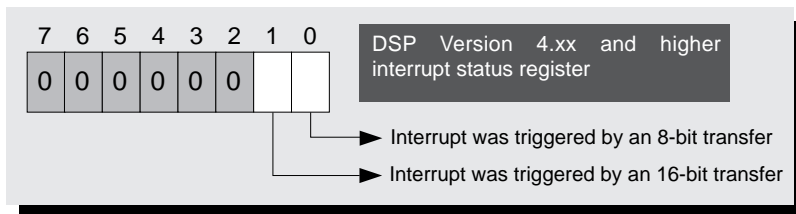
Beginning with DSP Version 4.xx, you must first check the source of the interrupt. Unlike earlier versions, completed 8-bit and 16-bit transfers can trigger interrupts with DSP Version 4.xx. Also, both 8-bit and 16-bit transfers share an interrupt and an interrupt handler. Therefore, the interrupt handler must first read register 82h of the Sound Blaster mixer, to find out what activity triggered the interrupt.

Like the OPL's internal registers, the mixer registers are addressed through separate address and data ports. The number of the desired mixer register is written to the address port; then its contents can be read or data written to that register through the data port.

The mixer's address and data ports are relative to the base address of the Sound Blaster card as follows:

```
Mixer address port = Sound Blaster base address + 4
Mixer data port = Sound Blaster base address + 5
```

Writing a value of 82h to the mixer address port and then reading the mixer data port, returns the contents of the Sound Blaster card's interrupt status register. The contents is as follows (found only on Sound Blaster cards with DSP Version 4.xx and above):



To acknowledge that an 8-bit transfer, the interrupt handler should read the port at offset 0Eh. To acknowledge a 16-bit transfer interrupt, the interrupt handler should read the port at offset 0Fh. This is the only way the DSP receives confirmation the interrupt was processed by the interrupt handler.

The following assembly language code demonstrates the confirmation by the interrupt handler:

```

;--- first read mixer register 82h -----
;---(Interrupt Status Register)
    mov     dx,SbPortBase    ;load port address of SB card
    add     dl,4             ;Offset for mixer address port
    mov     al,82h           ;identify the interrupt status register
    out     dx,al            ;write to mixer address port
    inc     dl               ;DX now at mixer data port
    in      al,dx            ;read contents of interrupt status register
                                ;from mixer data port

    add     dl,9             ;DX now at Read Buffer status reg.
    test    al,2             ;test whether 16-bit interrupt
    jne     SbInt16          ;set, then 16-bit

;--- it's an 8-bit interrupt
    in      al,dx            ;read Read Buffer status register,
                                ;to confirm to DSP

```

```

        jmp      Irqendwo      ;continue

;--- it's a 16-bit interrupt
SbInt16: inc      d            ;DX on/to register with offset 0Fh
        inal,dx              ;read register to confirm to DSP

Irqendwo:                      ; continue IRQ processing

```

The Sound Blaster interrupt is a hardware interrupt. Therefore, the interrupt handler will have to send an EOI command to the interrupt controller before returning to the interrupted program, i.e., before executing the IRET instruction. Otherwise, subsequent interrupts will be blocked. For more information on interrupts and interrupt controller, refer to Chapter 5.

DSP Commands

Most of the 40 DSP commands date back to Version 1.0. Half of these commands read and write the sample data. The other half perform housekeeping tasks such as switching on and off the DSP-amplifier connection or returning the DSP version number.

The following table summarizes these commands. Following the tables, are practical examples of some of the most important commands. Following the examples is a command reference describing all the commands in detail.

Listing of DSP commands by function group

The following commands are available beginning with DSP Version 1.xx.

Command type	Code	Command
8-bit direct mode commands	10h	output an 8-bit sample in direct mode
	20h	input an 8-bit sample in direct mode
Set sample frequency	40h	Set sample frequency
8-bit single cycle DMA mode commands	14h	8-bit single cycle DMA output
	24h	8-bit single cycle DMA input
	16h	8/2 bit ADPCM single cycle DMA output
	17h	8/2 bit ADPCM single cycle DMA output with reference byte
	74h	8/4 bit ADPCM single cycle DMA output
	75h	8/4 bit ADPCM single cycle DMA output with reference byte
	76h	8/3 bit ADPCM single cycle DMA output
	77h	8/3 bit ADPCM single cycle DMA output with reference byte
8-bit control commands	D0h	Pause 8-bit DMA transfer
	D4h	Continue 8-bit DMA transfer
Control speakers	D1h	Switch on speaker
	D3h	Switch off speaker
Miscellaneous	80h	Pause input briefly
	E1h	Get version number of DSP

The following commands are available beginning with DSP Version 2.0.

Command type	Code	Command
8-bit auto init DMA mode commands	1Ch	8-bit auto init DMA output
	2Ch	8-bit auto init DMA input
	1Fh	8/2 bit ADPCM auto init DMA output with reference byte
	7Dh	8/4 bit ADPCM auto init DMA output with reference byte
	7Fh	8/3 bit ADPCM auto init DMA output with reference byte
	DAh	Terminate 8-bit sample auto init DMA transfer
Set block transfer size	48h	Set block length for auto init and high speed DMA transfers
Control speakers	D8h	Get status of speaker

The following commands are available for DSP Version 2.01+ and 3.xx only.

Command type	Code	Command
8-bit high speed mode commands	90h	8-bit high speed auto init DMA output
	98h	8-bit high speed auto init DMA input
	91h	8-bit high speed single cycle DMA output
	99h	8-bit high speed single cycle DMA input

The following commands are only available for DSPs of Version 3.xx only:

Command type	Code	Command
Set Mono/Stereo input	A0h	Set input mode to Mono
	A8h	Set input mode to Stereo

The following commands are available for DSP Version 4.xx and higher.

Command type	Code	Command
Set sampling frequency	41h	Set sampling frequency for output
	42h	Set sampling frequency for input
Input and output 8-bit samples	Cxh	8-bit samples DMA input/output
Input and output 16-bit samples	Bxh	16-bit samples DMA input/output
	D5h	Pause 16-bit sample DMA transfer
	D6h	Continue 16-bit sample DMA transfer
	D9h	Terminate 16-bit sample auto init DMA transfer

Determining the DSP version

The E1h command is used to determine the version number of a DSP. To do this, read the Read Data register twice in succession after writing this command code. The DSP main version number is returned as the first byte and the DSP subversion number is returned as the second byte. The version number helps you determine which type of Sound Blaster card is installed.

Setting the sampling rate/sampling frequency

Before sampling begins, you must first set the sampling rate which is a function of the sampling frequency. To set the sampling rate, use DSP command 40h. In addition to the command code, you must also pass the desired sampling rate to the DSP. The rate is set according to the following formula:

$$\text{rate} = 65536 - (256000000 / (\text{Channels} * \text{Sampling frequency}))$$

where Channels = 1 for mono sampling *or*

Channels = 2 for stereo sampling

The most significant byte of the result is passed to the DSP; the least significant byte is discarded.

For DSP Versions 4.xx and higher, there are two additional commands for setting the sampling rate. These commands 41h and 42h are easier to use since the argument does not have to be converted as above. Both commands 41h and 42h are identical in syntax but one is for capturing samples and the other for playback of samples. Command 41h is used to set the sampling frequency for input (capture); command 42h is used for output (playback). With both commands, write the desired sample frequency in hertz to the Write command data port: first the most significant byte and then the least significant byte. It doesn't matter whether you are sampling in mono or stereo.

Setting the transfer length

Besides the sampling rate, you must also specify the transfer length so the DSP knows how many samples to capture or playback before triggering an interrupt. Depending on the type of sampling mode, set the transfer length separately using command 48h (for all auto init and high speed DMA modes) or specify it as part of the sampling command itself (for all single cycle DMA modes). After writing command code 48h, you then write the least followed by the most significant byte of the transfer length. Recall the DSP expects the transfer length less one. To transfer 4K (1000h), specify a length of 0FFFh.

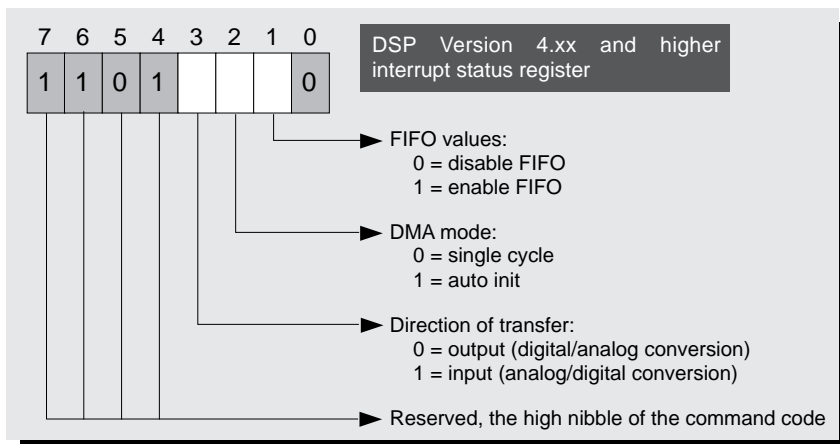
Switching the speaker on and off

Switching the speakers on or off refers to the connection between the DSP and the Sound Blaster amplifier. The speaker is switched on (DSP command D1h) to playback sample data or switched off (DSP command D3h) to capture sample data.

DSP Version 4.xx Commands

With DSP Version 4.xx, the Cxh and Bxh commands are used to capture and playback samples. The Cxh commands are for 8-bit samples and the Bxh commands for 16-bit samples. The x in both commands is a variable parameter; the lower nibble of the command code is defined by the desired mode of operation.

First, we'll have a look at Bxh. The complete command code is composed of a bit field whose upper four bits are always Bh, and whose lower four bits define a specific operation. The following illustration defines these operations:

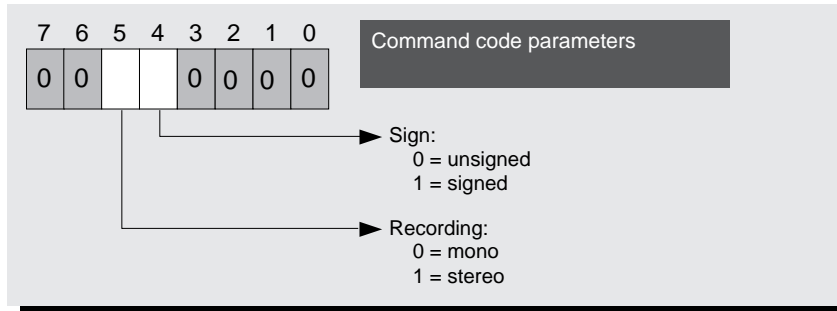


Bit 3 specifies either input (capture) or output (playback) operation. Bit 2 specifies whether single cycle or auto init DMA mode is used. Bit 1 specifies whether FIFO (a small internal buffer of the DSP) is disabled or enabled. With high sampling frequencies, you should enable FIFO; otherwise, when a higher priority DMA channel interferes with the DMA controller during data transfer, you can easily lose sample data.

For example, BAh (10111010b) specifies capturing 16-bit samples in single cycle mode with the FIFO enabled. B4h specifies playing back 16-bit samples in auto init mode without FIFO disabled.

To start an operation, first write the command code to the DSP. A second byte is then written containing two other parameters: the recording format (unsigned or signed) and the number of channels (mono or stereo).

The following illustration shows the structure of this byte:



Finally a 16-bit value is written that specifies the number of 16-bit samples (not bytes) -1. The value is sent least significant followed by most significant byte. The DSP command Cxh is the same except that it applies to 8-bit samples. The command code for Cxh is determined by the same three bit fields describing the Bxh command. However, the last 16-bit parameter specifies the number of 8-bit samples.

Before recording or playing back samples, first set the sampling rate. Use DSP command 41h to play back samples or command 42h to capture samples. With both commands, a 16-bit value follows the command code, specifying the sample frequency in Hertz. Values between 5,000 and 44,100 are permissible. It doesn't matter whether you are sampling in mono or stereo. Send the most significant byte of the sampling rate to the DSP first, followed by the least significant byte.

Before sampling, program the DMA controller with the starting address and number of sample bytes (Cxh) or sample words (Bx). Remember that 16-bit samples use a different DMA channel than 8-bit samples. After the desired number of samples have been captured or played, the DSP triggers an interrupt. If auto init DMA mode was selected, you can terminate this mode within the interrupt routine by using the DAh command for 8-bit samples and D9h for 16-bit samples.

8-bit Mono, Single Cycle DMA mode

Eight different commands are available for capturing and playing back 8-bit sample in single cycle DMA mode. They're all programmed similarly, so we'll describe them together. First, switch on the speaker. Command D1h switches the speaker on so you can playback the samples. Command D3h switches the speaker off so you can capture samples. Next set the sample frequency, using command 40h.

Now program the DMA controller by setting the starting memory address and block length. An interrupt handler must be activated to deal with the transferred data. To begin recording or playback of the samples, use one of the commands from the table followed by the transfer length parameter. This parameter is sent as two bytes: the least followed by most significant byte. Following the transfer, the DSP triggers an interrupt.

DSP commands for 8-bit Mono, Single Cycle DMA mode	
Command Code	Task
14h	8-bit single cycle DMA output
24h	8-bit single cycle DMA input
16h	8/2 bit ADPCM single cycle DMA output
17h	8/2 bit ADPCM single cycle DMA output with reference byte
74h	8/4 bit ADPCM single cycle DMA output
75h	8/4 bit ADPCM single cycle DMA output with reference byte
76h	8/3 bit ADPCM single cycle DMA output
77h	8/3 bit ADPCM single cycle DMA output with reference byte

After you've completed a recording or playback, you must disconnect the DSP from amplifier using command D3h. Now the operation is considered finished.

8-bit Mono, Auto Init DMA mode

Five different commands are available for capturing and playing back 8-bit samples in auto init DMA mode. These commands are similar to single cycle DMA mode except that a separate command is used to terminate the operation.

DSP commands for 8-bit Mono, Auto Init DMA mode	
Command Code	Task
1Ch	8-bit auto init DMA output
1Fh	8/2 bit ADPCM auto init DMA output with reference byte
2Ch	8-bit auto init DMA input
7Dh	8/4 bit ADPCM auto init DMA output with reference byte
7Fh	8/3 bit ADPCM auto init DMA output with reference byte

8-bit mono, High Speed Single Cycle DMA mode

DSP Version 3.xx features a high speed DMA mode to allow higher sampling rates. Two additional commands are used to record and play back 8-bit samples in the single cycle version of this mode.

DSP commands for 8-bit Mono, Auto Init DMA mode	
Command Code	Task
91h	8-bit high speed single cycle DMA output
99h	8-bit high speed single cycle DMA input

These two commands are similar to commands 1Ch and 2Ch. However, when high speed mode is complete, you must reset the DSP. Only then will the DSP be able to accept new commands.

8-bit Mono, High Speed Auto Init DMA mode

DSP Version 3.xx also includes two commands for recording and playing back samples in high speed auto init DMA mode.

DSP commands for 8-bit Mono, Auto Init DMA mode	
Command code	Task
90h	8-bit high speed auto init DMA output
98h	8-bit high speed auto init DMA input

These two commands are similar to commands 1Ch and 2Ch. However, when high speed mode is complete, you must reset the DSP. Only then will the DSP be able to accept new commands.

8-bit Stereo, High Speed Single Cycle DMA mode

DSP Version 3.xx also features stereo recording and playback. While this is a desirable feature, it's also more time-consuming to program. Sampling is started using commands 91h and 99h, described above. However, both before and after you send the above commands, you must perform certain other operations, which we'll describe.

DSP commands for 8-bit Mono, Auto Init DMA mode	
Command code	Task
91h	8-bit high speed single cycle DMA output
99h	8-bit high speed single cycle DMA input

For recording, first disconnect the speakers using DSP command D3h. Then switch to DSP stereo mode using command A8h. For playback, connect the speaker using command D1h. Next set bit 2 in mixer register 0Eh. Now transfer one byte containing the value 80h. For this, first program the DMA controller for the address of the single byte and a transfer length of one, then issue the DSP single cycle transfer command 14h, also specifying 1 byte as the transfer length.

Additional preparations for recording and playback of the stereo samples must still be made. Set the DMA controller for the address of the transfer buffer and block length. Next set the sampling frequency using command 40h. Now back to the mixer.

For playback the mixer register is 0Eh and for recording the mixer register is 0Ch. Read and save the contents of the respective register. Then set bit 5 and rewrite the contents back to the register. This activates the input or output filter.

Finally, start the sampling operation by first setting the transfer length using command 48h followed by either command 99h (record) or 91h (playback). When the block has been processed, an interrupt is triggered. Then perform additional stereo sample operations from within the interrupt handler. Simply program the DMA controller and issue DSP command 99h or 91h. It isn't necessary to perform the other operations again, such as accessing the mixer. When the stereo operations are complete, you'll have to undo these settings. Restore the previous contents of mixer register 0Ch or 0Eh which you already saved. Switch the DSP back to mono mode. For recording, return the DSP to mono mode simply by issuing command A0h. For playback, clear bit 1 in mixer register 0Eh. Finally, disconnect the speaker to conclude the whole operation. You'll probably discover that recording or playing back stereo samples using DSP Version 3.xx requires much effort.

8-bit Stereo, High Speed Auto Init DMA mode

These commands are similar to the auto init DMA mode commands. The major difference is that a transfer using high speed auto init DMA mode can be terminated only by resetting the DSP. Otherwise, the procedure is every bit as complicated as the procedure for previous high speed single cycle DMA mode.

The following table lists the DSP commands.

DSP commands for 8-bit stereo, High Speed Auto Init DMA mode	
Command code	Task
90h	8-Bit High Speed Auto Init DMA output
98h	8-Bit High Speed Auto Init DMA input

DSP commands by command code		DSP commands by command code	
10h	8-bit sample Direct Mode output	80h	Pause input
14h	8-bit Single Cycle DMA output	90h	8-bit High Speed Auto Init DMA output
16h	8/2 bit ADPCM Single Cycle DMA output	91h	8-bit High Speed Single Cycle DMA output
17h	8/2 bit ADPCM Single Cycle DMA output with reference byte	98h	8-bit High Speed Auto Init DMA input
1Ch	8 bit Auto Init DMA output	99h	8-bit High Speed Single Cycle DMA input
1Fh	8/2 bit ADPCM Auto Init DMA output with reference byte	A0h	Set input mode to mono
20h	8-bit Direct Mode input	A8h	Set input mode to stereo
24h	8-bit Single Cycle DMA input	Bxh	16-bit DMA input/output
2Ch	8-bit Auto Init DMA input	Cxh	8-bit DMA input/output
40h	Set sample frequency	D0h	Pause 8-bit DMA transfer
41h	Set sample frequency for output	D1h	Switch speaker on
42h	Set sample frequency for input	D3h	Switch speaker off
48h	Set block length for Auto Init and High-Speed DMA transfers	D4h	Continue 8-bit DMA transfer
74h	8/4 Bit ADPCM Single Cycle DMA output	D5h	Pause 16-bit DMA transfer
75h	8/4 Bit ADPCM Single Cycle DMA output with reference byte	D6h	Continue 16-bit DMA transfer
76h	8/3 Bit ADPCM Single Cycle DMA output	D8h	Get status of speaker
77h	8/3 Bit ADPCM Single Cycle DMA output with reference byte	D9h	Conclude 16-bit Auto Init DMA transfer
7Dh	8/4 Bit ADPCM Auto-Init DMA output with reference byte	DAh	Conclude 8-bit Auto Init DMA transfer
7Fh	8/3 Bit ADPCM Auto-Init DMA output with reference byte	E1h	Get version number of DSP

DSP Commands in detail

10h 8-bit Direct Mode Output

Output	10h	Command code
	bSample	sample byte to be output

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Output an 8-bit sample through the DSP. The frequency at which this command is executed determines the sampling rate.

14h 8-bit Single Cycle DMA Output

Output	14h	Command code
	lo(Length)	Low-byte of block length - 1
	hi(Length)	High-byte of block length - 1

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Output a block of 8-bit samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and block length. When using this command, specify block length - 1.

16h 8/2 bit ADPCM Single Cycle DMA Output

Output	16h	Command code
	lo(Length)	Low-byte of block length - 1
	hi(Length)	High-byte of block length - 1

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Output a block of 8/2 bit ADPCM samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and block length. When using this command, specify the block length - 1. This command is used to output the second and all subsequent 8/2 bit ADPCM sample blocks. Output the first block using command 17h.

17h 8/2 bit ADPCM Single Cycle DMA Output with reference byte

Output	17h	Command code
	lo(Length-1)	Low-byte of block length - 1
	hi(Length-1)	High-byte of block length - 1

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Output a block of 8/2 bit ADPCM samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and the block length. The first byte of the block is a normal 8-bit sample, not an ADPCM compressed sample. Subsequent blocks are output using command 16h.

1Ch 8-bit Auto Init DMA Output

Output	1Ch	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
	●	●	●	●

Output a block of 8-bit samples using Auto Init DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and block length and set the DSP block length using command 48h.

Two commands can be used to terminate an Auto Init operation. Use either command DAh to terminate Auto Init mode or command 14h to return to Single Cycle mode. Use command D0h to pause Auto Init mode. Use command D4h to resume processing following the pause.

37. Sound Blaster And Compatible Cards

631

1Fh 8/2 bit ADPCM Auto Init DMA Output with reference byte

Output	1Fh	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
	●	●	●	●

Output a block of 8/2 bit ADPCM samples using Auto Init DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h. The first byte of the data block contains the reference byte

Two commands can be used to terminate an Auto Init operation. Use either command DAh to terminate Auto Init mode or command 16h to return to Single Cycle 8/2 bit ADPCM mode.

Use command D0h to pause Auto Init mode. Use command D4h to resume processing following the pause.

20h 8-bit Direct Mode Input

Output	20h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Input an 8-bit sample through the DSP. The sample byte is read from the DSP Read -Data port. The frequency at which this command is executed determines the sampling rate.

24h 8-bit Single Cycle DMA input

Output	24h	Command code
	lo(Length-1)	low-byte of block length - 1
	hi(Length-1)	high-byte of block length - 1

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Input a block of 8-bit samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and block length. When using this command, specify the block length - 1.

2Ch 8-bit Auto Init DMA input

Output	2Ch	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
	●	●	●	●

Input a block of 8-bit samples using Auto Init DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length, and set the DMA block length using command 48h.

Two commands can be used to terminate an Auto Init operation. Use either command DAh to terminate Auto Init mode or command 24h to return to Single Cycle DMA mode.

Use command D0h to pause Auto Init mode. Use command D4h to resume processing following the pause.

40h Set sample frequency

Output	40h	Command code
	hi(time constant)	most significant byte of time constant

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Use one of these formula is used to calculate the time constant for a desired *sample frequency*:

16 Bit: time constant = $65536 - (256000000 / (\text{Channels} * \text{Sample frequency}))$,
 8 Bit: time constant = $256 - (1000000 / (\text{Channels} * \text{Sample frequency}))$,

where *channels* = 1 for mono;

= 2 for stereo

Only the most significant (high)byte of the resulting value is passed to the DSP by this command. For DSP Version 4.xx, you can set the sample frequency using commands 41h and 42h.

41h

Set Output Sample Frequency

Output	41h	Command code
	hi(Sample frequency)	high-byte of sample frequency
	lo(Sample frequency)	low-byte of sample frequency

1.xx	2.0	2.01+	3.xx	4.xx
				●

This DSP 4.xx command sets the sample frequency for Single Cycle and Auto Init DMA output transfers. The sample frequency may range between 5,000 and 45,000 (Hertz). This command is for either mono or stereo modes.

42h

Set Input Sample Frequency

Output	41h	Command code
	hi(Sampling frequency)	high-byte of sampling frequency
	lo(Sampling frequency)	low-byte of sampling frequency

1.xx	2.0	2.01+	3.xx	4.xx
				●

This DSP 4.xx command sets the sample frequency for Single Cycle and Auto Init DMA input transfers. The sample frequency may range between 5,000 and 45,000 (Hertz). This command is for either mono or stereo modes.

48h

Set Auto Init and High Speed DMA Block Length

Output	48h	Command code
	hi(BlockLength-1)	high-byte of block length in bytes - 1
	lo(BlockLength-1)	low-byte of block length in bytes - 1

1.xx	2.0	2.01+	3.xx	4.xx
	●	●	●	●

This command sets the block length in bytes for subsequent Auto Init and High Speed DMA transfers. An interrupt is triggered after this amount of data is transferred.

74h

8/4 bit ADPCM Single Cycle DMA Output

Output	74h	Command code
	lo(Length-1)	low-byte of block length - 1
	hi(Length-1)	high-byte of block length - 1

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Output a block of 8/4 bit ADPCM samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and block length. When using this command, specify block length - 1. This command is used to output the second and all subsequent 8/4 bit ADPCM sample blocks. Output the first block using command 75h.

75h

8/4 bit ADPCM Single Cycle DMA Output with reference byte

Output	75h	Command code
	lo(Length-1)	low-byte of block length - 1
	hi(Length-1)	high-byte of block length - 1

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

37. Sound Blaster And Compatible Cards

633

Output a block of 8/4 bit ADPCM samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and block length. The first byte of the block is a normal 8-bit sample, not an ADPCM compressed sample. Subsequent blocks are output using command 74h.

76h			8/3 bit ADPCM Single Cycle DMA Output				
Output	76h	Command code	1.xx	2.0	2.01+	3.xx	4.xx
	lo(Length-1)	low-byte of block length - 1	●	●	●	●	●
	hi(Length-1)	high-byte of block length - 1					

Output a block of 8/3 bit ADPCM samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and the block length. When using this command specify block length - 1. This command is used to output the second and all subsequent 8/3 bit ADPCM sample blocks. Output the first block using command 77h.

77h			8/3 bit ADPCM Single Cycle DMA Output with reference byte				
Output	77h	Command code	1.xx	2.0	2.01+	3.xx	4.xx
	lo(Length-1)	low-byte of block length - 1	●	●	●	●	●
	hi(Length-1)	high-byte of block length - 1					

Output a block of 8/3 bit ADPCM samples using Single Cycle DMA mode through the DSP. Before using this command, set the sample frequency and program the DMA controller with the starting address and the block length. The first byte of the block is a normal 8-bit sample, not an ADPCM compressed sample. Subsequent blocks are output using command 76h.

7Dh			8/4 bit ADPCM Auto Init DMA Output with reference byte				
Output	7Dh	Command code	1.xx	2.0	2.01+	3.xx	4.xx
				●	●	●	●

Output a block of 8/4 bit ADPCM samples to the DSP using Auto Init DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h. The first byte of the data block contains the reference byte

Two commands can be used to terminate an Auto Init operation. Use either command DAh to terminate Auto Init mode or command 74h to return to Single Cycle 8/4 bit ADPCM output mode. Use command D0h to pause Auto Init mode. Use command D4h to resume processing following the pause

7Fh			8/3 bit ADPCM Auto Init DMA Output with reference byte				
Output	7Fh	Command code	1.xx	2.0	2.01+	3.xx	4.xx
				●	●	●	●

Output a block of 8/3 bit ADPCM samples using Auto Init DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length, and set the DSP block length using command 48h. The first byte of the data block contains the reference byte

Two commands can be used to terminate an Auto Init operation. Use either command DAh to terminate Auto Init mode or command 76h to return to Single Cycle 8/3 bit ADPCM output mode. Use command D0h to pause Auto Init mode. Use command D4h to resume processing following the pause.

80h Pause Input

Output	80h	Command code
	lo(PauseDuration - 1)	low-byte of duration of pause - 1
	hi(PauseDuration - 1)	high-byte of duration of pause - 1

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Temporarily pause (halt) the transfer started using Single Cycle or Auto Init DMA input. The duration of the pause is relative to the sample frequency set by command 40h. After this period of time has passed, the transfer resumes.

90h 8-bit High Speed Auto Init DMA Output

Output	90h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
		●	●	

Output a block of 8-bit samples using High Speed Auto Init DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h.

To terminate High Speed Auto Init mode, you must reset the DSP.

91h 8-bit High Speed Single Cycle DMA Output

Output	91h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
		●	●	

Output a block of 8-bit samples using Single Cycle High Speed DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h.

98h 8-bit High Speed Auto Init DMA Input

Output	98h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
		●	●	

Input a block of 8-bit samples in High Speed Auto Init DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h

To terminate High Speed Auto Init mode, you must reset the DSP.

99h 8-bit High Speed Single Cycle DMA Input

Output	91h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
		●	●	

Input a block of 8-bit samples using Single Cycle High Speed DMA mode through the DSP. Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h

A0h Set Input mode to Mono

Output	A0h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
			●	

Set the DSP input mode to mono. This is the default setting (see below).

37. Sound Blaster And Compatible Cards

635

A8h Set Input mode to Stereo

Output	A8h	Command code
--------	-----	--------------

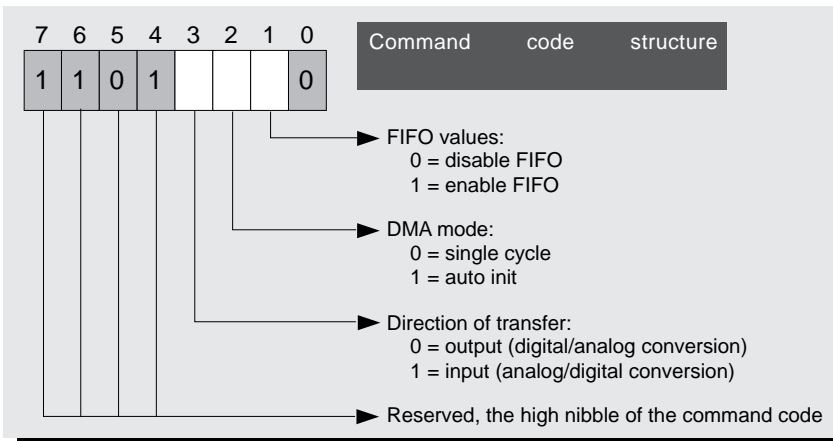
1.xx	2.0	2.01+	3.xx	4.xx
			●	

Set the DSP input mode to stereo. Reset this mode to mono when recording is complete. Since this command is not available after DSP Version 4.xx, use command Bxh instead.

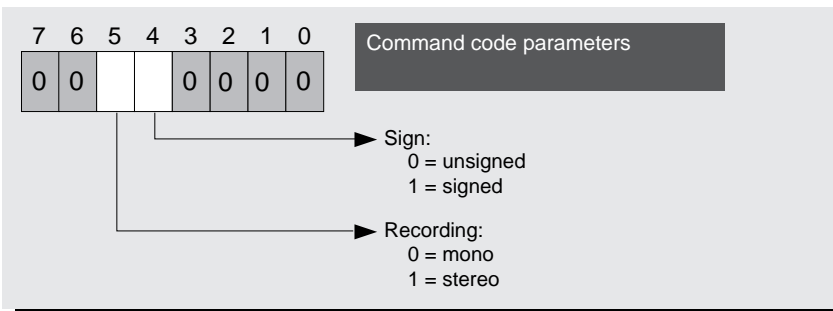
Bxh 16-bit DMA Input/Output

Output	Bxh	Command code
	Operation mode	see below
	lo(Samples-1)	low-byte of number of 16-bit samples - 1
	hi(Samples-1)	high-byte of number of 16-bit samples - 1

For DSP Version 4.xx, input or output a 16-bit sample block. The complete command code varies depending on the transfer direction, the selected DMA mode and usage of the FIFO buffer. The command code has the following structure:



The operation mode contains two bit fields that specify whether to sample in mono or stereo and also whether the samples are unsigned or signed. Using unsigned values, the smallest measurable signal amplitude corresponds to the value 0h; using signed values, this corresponds to 8000h (-1).



Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h. For an Auto Init DMA transfer, you can terminate this mode by using

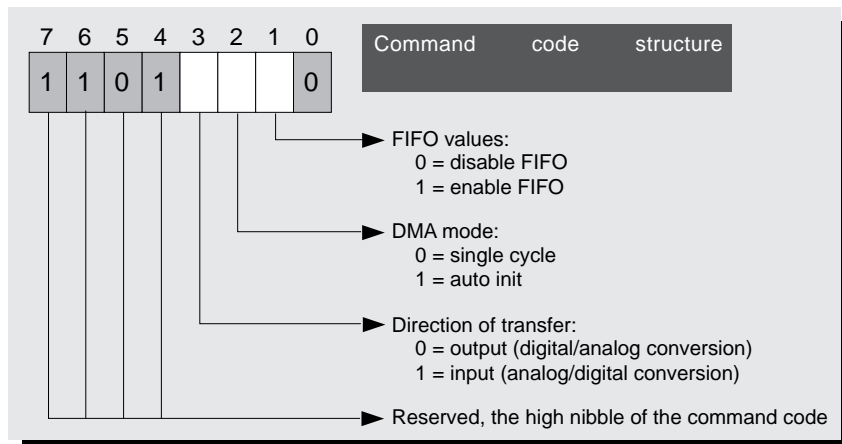
command D9h to terminate Auto Init mode or command Bxh again with bit 2=0 (Single Cycle). Use command D5h to pause Auto Init mode. Use command D6h to resume processing following the pause.

1.xx	2.0	2.01+	3.xx	4.xx
				●

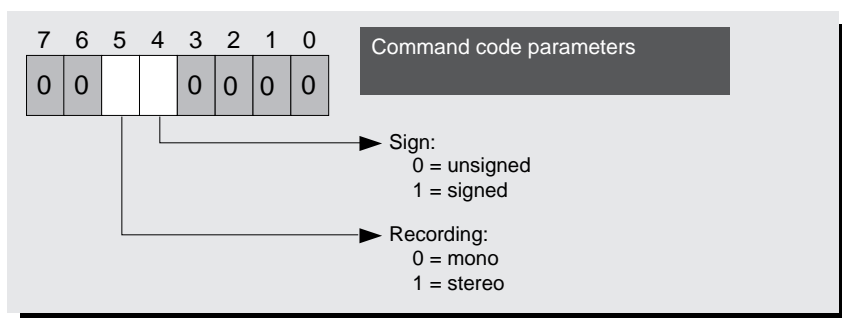
Cxh 8-bit DMA Input/Output

Output	Cxh	Command code
	Operation mode	see below
	lo(Samples-1)	low-byte of number of 8-bit samples - 1
	hi(Samples-1)	high-byte of number of 8-bit samples - 1

For DSP Version 4.xx, input or output an 8-bit sample block. The complete command code varies depending on the transfer direction, the selected DMA mode and usage of the FIFO buffer. The command code has the following structure:



The operation mode contains two bit fields that specify whether to sample in mono or stereo and also whether the samples are unsigned or signed. Using unsigned values, the smallest measurable signal amplitude corresponds to the value 0h; using signed values, this corresponds to 80h (-1).



Before using this command, set the sample frequency, program the DMA controller with the starting address and block length and set the DSP block length using command 48h. For an Auto Init DMA transfer, you can terminate this mode by using command DAh to terminate Auto Init mode or command Cxh again with bit 2=0 (Single Cycle).

1.xx	2.0	2.01+	3.xx	4.xx
				●

Use command D0h to pause Auto Init mode. Use command D4h to resume processing following the pause.

37. Sound Blaster And Compatible Cards

637

D0h 8-bit DMA Input/Output

Output	D0h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Temporarily pause an 8-bit DMA transfer started in Single Cycle or Auto Init DMA mode. Command D4h resumes the transfer.

D1h Switch speaker on

Output	D1h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Connect the DSP to the amplifier on the Sound Blaster card. This command is discontinued, starting with DSP Version 4.xx.

D3h Switch speaker off

Output	D1h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Disconnect the DSP from the amplifier on the Sound Blaster card. This command is discontinued, starting with DSP Version 4.xx.

D4h Resume 8-bit DMA transfer

Output	D4h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Resume the 8-bit DMA transfer that has been paused by command D0h

D5h Pause 16-bit sample DMA transfer

Output	D5h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
				●

Temporarily pause (halt) a 16-bit DMA transfer started by the Bxh command. Use command D6h to resume the transfer.

D6h Resume 16-bit sample DMA transfer

Output	D5h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
				●

Resume the 16-bit DMA transfer that has been paused by command D5h

D8h Get speaker status

Output	D8h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
	●	●	●	●

Returns the status of the DSP connection to the Sound Blaster card amplifier. After writing this command code, read the Read-Data port. A value of 0FFh (-1) means they are connected; a value of zero means they are not connected.

D9h Terminate 16-bit Auto Init DMA transfer

Output	D9h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
				●

Terminate the 16-bit Auto Init DMA transfer initiated by command Bxh. The current transfer is allowed to complete before the operation is terminated.

DAh

Terminate 8-bit Auto Init DMA transfer

Output	DAh	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
				●

Terminate the 8-bit Auto Init DMA transfer initiated by command Cxh. The current transfer is allowed to complete before the operation is terminated.

E1h

Get DSP Version number

Output	E1h	Command code
--------	-----	--------------

1.xx	2.0	2.01+	3.xx	4.xx
●	●	●	●	●

Returns the DSP version number. After writing this command code, read the Read-Data register twice. The first byte is the DSP main version number; the second byte is the DSP subversion number. Use the version number to determine what kind of Sound Blaster card is installed.

You'll find the following program(s) on the companion CD-ROM



DSPUTIL.H (C listing)
 DSP.C (C listing)
 ARGS.PAS (Pascal listing)
 SBUTIL.PAS (Pascal listing)
 MIXUTIL.PAS (Pascal listing)
 DSP.PAS (Pascal listing)
 DSPUTIL.PAS (Pascal listing)
 DMAUTIL.PAS (Pascal listing)

The Mixer

The mixer controls the volume of the individual input and output sources, specifies how the sources are to be connected and enables the sample input signals.

Since different DSPs are used on various Sound Blaster cards, the mixers used on these cards in the Sound Blaster family are also different. There are three types of mixers:

1. CT1335 - This is a relatively primitive mixer; used primarily on the Sound Blaster 2.0 with CD-ROM drive expansion board.
2. CT1345 - A more advanced mixer; found on Sound Blaster Pro cards.
3. CT1745 - A more powerful mixer; introduced on Sound Blaster 16.

The features of the mixers vary from primitive to powerful. The mixer settings affect the combination of different inputs and outputs, the provision of filters for noise reduction and the sound card's resolution capacity relative to the volume of a sound source or an output signal. The greater the resolution capacity, the more different input levels are possible for the different sources.

Controlling the mixer

The mixer registers are addressed through separate address and data ports. The number of the desired mixer register is written to the address port, then its contents can be read from or data written to that register through the data port. The mixer's address and data ports are at offsets to the Sound Blaster card base address as follows:

Mixer Address Port = Sound Blaster Base Address + 4
 Mixer Data Port = Sound Blaster Base Address + 5

Unlike the OPL chip, after writing the desired register number to the mixer address port, you can immediately read from or write to the register.

This assembly language code shows you how to write to one of the mixer's registers:

```

mov    dx,SbPortBasis    ;Load port address of SB card
add    dl,04h            ;Offset for mixer address port
mov    al,MixRegNr       ;Number of desired mixer register
out    dx,a              ;Send to mixer address port
inc    d                 ;Set access to mixer data port
mov    al,NewWord        ;load new value for register
out    dx,A              ;and write to mixer data port

```

The following shows you how to read the contents of one of the mixer's registers:

```

mov    dx,SbPortBasis    ;Load port address of SB card
add    dl,04h            ;Offset of mixer address port
mov    al,MixRegNr       ;Number of desired mixer register
out    dx,a              ;Send to mixer address port
inc    d                 ;Set access to mixer data port
in     al,dx             ;and read mixer registers

```

In the following descriptions, you'll notice the function of many of the register bits is undefined. Creative Labs warns you to not make assumptions about the contents of the reserved bits when accessing these registers and suggests the following procedures:

1. To test the contents of a register: read and then mask out the undefined bits.
- 2.. To modify the contents of a register: read the register contents; set or reset only those bits that require changing; rewrite the entire register contents.
3. To protect the mixer environment: a program that changes the mixer registers should initially read and save the register contents and later restore the register contents before ending

CT1335 mixer

The CT1335 mixer has five registers. These registers control the various output sources. It also lets you set the volume for Line-In, MIDI, CD and Microphone.

The CT1335 Mixer

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00h	Mixer-Reset							
02h					Master-Volume			
06h					MIDI-Volume			
08h					CD-Volume			
0Ah					Voice-Volume			

Register 00h

Mixer Reset

To reset the mixer, write any value to this register. Reset restores the default register contents.

Register 02h

Master Volume

To set a master volume level, write a value from 0 to 7 to this register. This value changes the volume level from -46 to 0 decibels in increments of about 4 decibels. The default is 4, for -11 decibels.

Register 06h MIDI Volume

Same as register 02h, but for MIDI.

Register 08h CD Volume

Same as register 02, but for CD except the default is 0, for -46 decibels.

Register 0Ah Voice Volume

To set a voice volume, write a value from 0 to 3 to this register. This value changes the volume level from -46 decibels to 0 decibels in increments of about 7 decibels. The default is 0, for -46 decibels.

CT1345 Mixer

The CT1345 mixer has separate volume controls for the left and right stereo channels and lets you select the input source to feed to the DSP for sampling. It also contains an input and output filter for noise suppression.

The 1345 Mixer

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0			
00h	Mixer-Reset										
04h	Voice-Volume left				Voice--Volume right						
0Ah						Microphone Volume					
0Ch						Input-Filter		Low-Pass-Filter	Sample source		
0Eh						Output-Filter				Stereo-Switch	
22h					Master-Volume left				Master-Volume right		
26h	FM--Volume left				FM-Volume right						
28h	CD--Volume left				CD-Volume right						
2Eh	Line--Volume left				Line-Volume right						

Register 00h Mixer Reset

To reset the mixer, write any value to this register. Reset restores the default register contents.

Register 04h Voice Volume Left & Right (Values 0-15)

Register 0Ah, Bits 0-2 Microphone Volume

To set a microphone volume level, write a value from 0 to 7 to this register. This value changes the volume level from -46 to 0 decibels in increments of about 6 decibels. The default is 0, for -46 decibels.

Register 0Ch, Bits 1-2 Input Source

00b	Microphone (Default)	01b	CD
10b	Microphone	11b	Line In

Register 0Ch, Bit 3 Set Input Filter

Specifies the range of the input filter, if register 0Ch, Bit 5 is set. If register 0Eh, Bit 5 is set, it becomes an output filter.

0b	3.2 Khz Low Pass Filter (Default)	1b	8.8 Khz Low Pass Filter (High Filter)
----	-----------------------------------	----	---------------------------------------

Register 0Ch, Bit 5	Activate Input Filter
---------------------	-----------------------

0b	Low Pass Input Filter is active (Default)	1b	Low Pass Input Filter is inactive
----	---	----	-----------------------------------

Register 0Eh, Bit 1	Stereo Switch
---------------------	---------------

Enable mono or stereo output.

0b	Mono output (Default)	1b	Stereo output
----	-----------------------	----	---------------

Register 0Eh, Bit 5	Activate Output Filter
---------------------	------------------------

0b	Low Pass Output Filter is active (Default)	1b	Low Pass Output Filter is inactive
----	--	----	------------------------------------

Register 22h	Master Volume Left & Right (Values 0-15)
--------------	--

Register 26h	FM Volume Left & Right
--------------	------------------------

To set the Master or FM volume level, write a value from 0 to 15 to one of these registers. This value changes the volume level from -46 to 0 decibels in increments of about 4 decibels. The default is 4, for -11 decibels.

Register 28h	CD Volume Left & Right (Values 0-15)
--------------	--------------------------------------

Register 2Eh	Line Volume Left & Right (Values 0-15)
--------------	--

To set the CD or Line volume level, write a value from 0 to 15 to one of these registers. This value changes the volume level from -46 to 0 decibels in increments of about 4 decibels. The default is 0, for -46 decibels.

The low pass filters are used to attenuate high-frequency sounds during recording to achieve a better recording quality. The 3.2 KHz filter is recommended for mono samples with sampling rates less than 18 KHz, while the 8.8 KHz filter is recommended for mono samples with sampling rates between 18 KHz and 36 KHz.

Disable both filters for all stereo sampling and for mono sampling at a sampling rate greater than 36 KHz.

CT1745 Mixer

The CT1745 mixer is the top of the line mixer in the Sound Blaster line. The large number of registers is an indication of this mixer's capabilities.

The CT1745 has any new features. For example, the resolution of the individual volume registers is finer. Several output sources can be mixed into the output signal. Several input sources can be mixed into the input signal. Finally, the treble and bass settings are individually adjustable for the left and right output channels. The CT1745 lacks the filter bits of the CT1345, since the new mixer uses dynamic signal filtering.

*The CT1745
Mixer*

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0				
00h	Mixer-Reset											
04h	Voice-Volume left				Voice-Volume right							
0Ah						Microphone--Volume						
22h	Master-Volume left				Master-Volume right							
26h	FM-Volume left				FM-Volume right							
28h	CD-Volume left				CD-Volume right							
2Eh	Line-Volume left				Line-Volume right							
30h												
31h									Master-Volume right			
32h									Voice-Volume left			
33h									Voice-Volume right			
34h									MIDI-Volume left			
35h									MIDI-Volume right			
36h									CD-Volume left			
37h									CD-Volume right			
38h									Line-Volume left			
39h									Line-Volume right			
3Ah									Microphone-Volume			
3Bh									PC-Loudspeaker Volume			
3Ch				Line left	Line right	CD left	CD right	Microphone				
3Dh				MIDI left	MIDI right	Line left	Line right	CD left	CD right	Microphone		
3Eh				MIDI left	MIDI right	Line left	Line right	CD left	CD right	Microphone		
3Fh	Input Preamplifier left											
40h	Input Preamplifier right											
41h	Output Preamplifier left											
42h	Output Preamplifier right											
43h												
44h	Treble left				AGC							
45h	Treble right											
46h	Bass left											
47h	Bass right											

Register 00h Mixer Reset

To reset the mixer, write any value to this register. Reset restores the default register contents.

Register 04h Voice Volume Left & Right

Register 0Ah Microphone Volume

To set the Microphone volume level, write a value from 0 to 7 to one of this register. This value changes the volume level from -42 to 0 decibels in increments of about 6 decibels. The default is 0, for -42 decibels.

Register 0Ah is for compatibility with the previous mixers. You can also set the microphone volume using register 3Ah which offers finer resolution.

Register 22h Master Volume Left & Right

Register 26h FM Volume Left & Right

To set the Master or FM volume level, write a value from 0 to 15 to one of these registers. This value changes the volume level from -60 to 0 decibels in increments of about 4 decibels. The default is 12, for -12 decibels.

37. Sound Blaster And Compatible Cards

643

Registers 22h and 26h are for compatibility with the previous mixers. You can also set the volumes using registers 30h to 3Ah which offers finer resolution.

Register 28h	CD Volume Left & Right
--------------	------------------------

Register 2Eh	Line Volume Left & Right
--------------	--------------------------

To set the CD and Line volume levels, write a value from 0 to 15 to these register. This value changes the volume level from -60 to 0 decibels in increments of about 4 decibels. The default is 0, for -60 decibels.

Registers 28H and 2Eh are for compatibility with the previous mixers. You can also set the line volume using registers 30h to 3Ah which offers finer resolution.

Register 30h/31h	FM Volume Left & Right
------------------	------------------------

Register 32h/33h	Voice Volume Left & Right
------------------	---------------------------

Register 34h/35h	MIDI Volume Left & Right
------------------	--------------------------

To set any of these volume levels, write a value from 0 to 31 to these register. This value changes the volume level from -62 to 0 decibels in increments of about 2 decibels. The default is 24, for -14 decibels.

Register 36h/37h	CD Volume Left & Right
------------------	------------------------

Register 38h/39h	Line Volume Left & Right
------------------	--------------------------

Register 3Ah	Microphone Volume Left & Right
--------------	--------------------------------

To set any of these volume levels, write a value from 0 to 31 to these register. This value changes the volume level from -62 to 0 decibels in increments of about 2 decibels. The default is 24, for -14 decibels.

Register 3Bh	PC Speaker Volume
--------------	-------------------

To set the speaker volume level, write a value from 0 to 3 to this register. This value changes the volume level from -18 to 0 decibels in increments of about 6 decibels. The default is 0, for -18 decibels.

Register 3Ch	Output Mixer
--------------	--------------

To connect a source to the output signal, set the respective bit.

Register 3Dh	Left Channel Input Mixer
--------------	--------------------------

To connect a source to the left input signal, set the respective bit.

Mono recordings are sampled from this channel. For mono recordings from a stereo device, map both channels to this left input channel using registers 3Dh and 3Eh.

Register 3Eh	Right Channel Input Mixer
--------------	---------------------------

To connect a source to the right input signal, set the respective bit.

Register 3Fh/40h	Input Preamplifier Left & Right
------------------	---------------------------------

Register 41h/42h	Output Preamplifier Left & Right
------------------	----------------------------------

To set the preamplifier volume levels, write a value from 0 to 3 to one of these registers. This value changes the volume level from -18 to 0 decibels in increments of about 6 decibels. The default is 0, for 0 decibels.

Register 43h, Bit 0	Automatic Microphone Preamplifier (AGC)
0 = AGC on	1 = Preamplifier fixed at 20 dB
Register 44h/45h	Treble Left & Right
Register 46h/47h	Bass Left & Right

To set the treble or bass levels, write a value from 0 to 15 to one of these registers. This value changes the level from -14 to +14 decibels in increments of about 2 decibels. The default is 8, for 0 decibels.

Programming the mixers

In the following table, module MIXUTIL contains functions to query and set the mixer registers. As you read through following table, you'll see that we've differentiated between functions for accessing the CT1335 mixer (mix_...), the CT1345 mixer (mix3_...) and the CT1745 mixer (mix4_...).

MIXUTIL functions	
Function	Task
mix_SetBase	Set port address of Sound Blaster card
mix_Write	Send byte to mixer register
mix_Read	Read byte from mixer register
mix_Reset	Reset mixer
mix3_SetADCFilter	Switch ADC filter on or off
mix3_GetADCFilter	Get status of ADC filter
mix3_SetDACFilter	Switch DAC filter on or off
mix3_GetDACFilter	Get status of DAC filter
...	
...	
mix3_SetDACStereo	Switch stereo playback (DAC) on or off
mix3_GetDACStereo	Get status of stereo playback
mix3_SetADDACLowPass	Set low pass filter for recording
mix3_GetADDACLowPass	Get low pass filter being used
mix3_PrepareForStereo	Mixer for stereo recording/playback
mix3_RestoreFromStereo	Restore mixer to old state
mix3_SetVolume	Set output volume
mix3_GetVolume	Get output volume
mix3_SetADCSource	Set recording source
mix3_GetADCSource	Get current recording source
mix4_PrepareForMonoADC	Prepare mixer for mono recording
mix4_RestoreFromMonoADC	Restore mixer state
mix4_SetVolume	Set output volume of a mixer source
mix4_GetVolume	Get output volume

MIXUTIL functions (continued)	
Function	Task
mix4_SetADCSrc	Set recording source for left ADC
mix4_SetADCSrcR	Set recording source for right ADC
mix4_GetADCSrc	Get recording source for left ADC
mix4_GetADCSrcR	Get recording source for right ADC
mix4_SetOUTSrc	Set output(playback) source
mix4_GetOUTSrc	Get output source
mix4_SetADCGain	Set recording preamplifier
mix4_GetADCGain	Get recording preamplifier settings
mix4_SetOUTGain	Set output preamplifier
mix4_GetOUTGain	Get output preamplifier settings
mix4_SetAGC	Set/clear automatic gain control
mix4_GetAGC	Get automatic gain control
mix4_SetTreble	Set treble
mix4_GetTreble	Get treble setting
mix4_SetBass	Set bass
mix4_GetBass	Get bass setting

MIX.C demonstrates the use of these functions. Run the MIX program using the appropriate switches to set the desired mixer registers. Run this program without a switch, MIXUTIL.C to display the syntax on screen.

You'll find the following program(s) on the companion CD-ROM



MIXUTIL.C (C listing)
 MIX.C (C listing)
 MIX.PAS (Pascal listing)
 SBUTIL.PAS (Pascal listing)
 DSPUTIL.PAS (Pascal listing)
 ARGS.PAS (Pascal listing)
 MIXUTIL.PAS (Pascal listing)
 IRQUTIL.PAS (Pascal listing)

Speech Output

A talking machine is a particularly fascinating idea to many people. Some still consider this to be pure "science fiction" but digitized speech has been around for several years. Two early computers, the Apple II and the Commodore 64, both had speech capabilities for converting text to speech. The Sound Blaster card too, has speech capabilities.

One problem in digitized speech output is the amount of disk space required for the data..

The quality of speech output is related to the number of recognizable phoneme sequences. After all, an A in front of an H sounds much different than an A following an S, and an S at the end of a word can easily turn into a Z ("Bees"), while at the beginning of a word it actually sounds like ESS. Quality speech output recognizes these differences.

In this section, you'll see that digitized speech from the Sound Blaster is easy to use.

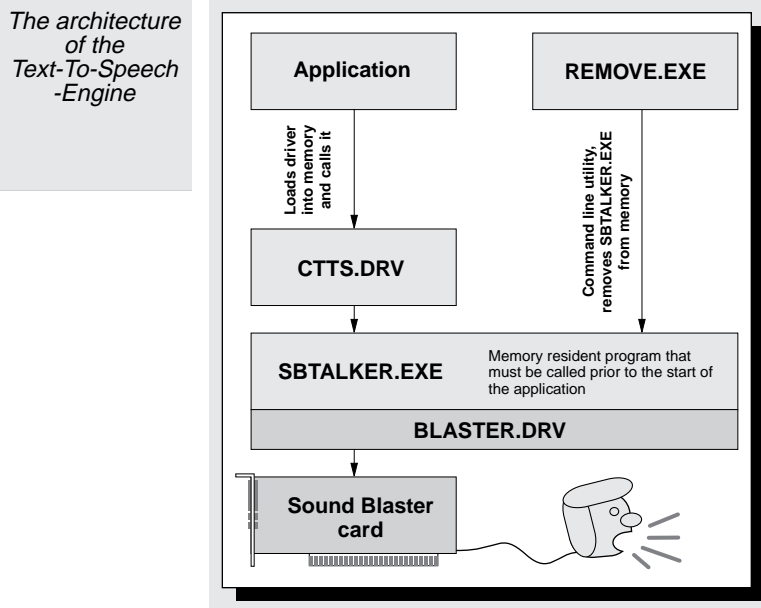
The architecture of the Text-To-Speech Engine

To use the Sound Blaster speech output, these drivers are needed: CTTS.DRV, SBTALKER.EXE and BLASTER.DRV. When you run the Sound Blaster install, these files are copied to your hard drive, so they should be ready to use. The following shows the role of these drivers in generating speech. SBTALKER, the actual speech module, is a TSR (terminate and stay resident) program. Usually, you run this program immediately before you are to perform any speech output. Alternatively, you can start this program from within another program (use the DOS EXEC function - see Chapter 25).

SBTALKER accesses the Sound Blaster card through driver BLASTER.DRV. This driver is in the same directory as SBTALKER (usually C:\SBTALKER). The installation program also places a batch file there named SBTALK.BAT which has only one line:

```
SBTALKER /dBLASTER
```

This line runs the SBTALKER program. The /d switch (d represents driver) is the name of the environment variable containing the Sound Blaster hardware settings. If you call SBTALKER from a program, don't forget to specify this /d switch.



Since SBTALKER is a TSR program that resides in memory, you may want to remove it from memory when you end your program. The REMOVE.EXE program in this same directory performs this function. SBTALKER contains all the functions necessary for speech output. However, another driver, CTTS.DRV links your talking program to SBTALKER. Your program loads this driver into memory before outputting speech. SBUTIL, already introduced to you, contains two functions for this purpose: sb_LoadDriver and sb_UnloadDriver. sb_LoadDriver requires the filename of the driver as its only argument. With this information, it determines the driver size, allocates a sufficient amount of RAM and loads the driver in the allocated memory segment. It also assures the driver begins at a paragraph boundary (an address that is divisible by 16), a prerequisite for smooth operation of the driver.

The following section describes the different driver functions. After the descriptions of the driver functions, we'll describe the CTTS module which contains high-level language functions for using all these driver functions.

Function 0

Get Version Number

Returns CTTS driver version number

Input	BX = 0	Output	AH = Main version number A = Subversion number
--------------	--------	---------------	---

Function 1	Get Blaster environment variable
-------------------	----------------------------------

Returns the contents of the Blaster environment variable containing the Sound Blaster hardware settings

Input	BX = 1 ES:DI = FAR pointer to an ASCIIZ string containing the Blaster environment variable. The string has to start after the equals sign, so the text "BLASTER=" does not appear in the string.	Output	AX = 0: OK 1: Error, ES:DI is NULL or pointing at a blank string 2: Error, invalid settings in string
--------------	---	---------------	---

This must be the first called function in a program that uses this driver.

Function 2	Initialize CTTS driver
-------------------	------------------------

Initialize the CTTS driver. This must be the second called function in a program that uses this driver.

Input	BX = 2	Output	AX = 0: OK <> 0: Error
--------------	--------	---------------	---------------------------

Before this function can be called, function 1 must be successfully called.

Function 3	End driver
-------------------	------------

Informs the driver that no additional driver functions will be called.

Input	BX = 3	Output	None
--------------	--------	---------------	------

This function does not remove itself from memory. To call other driver functions afterwards, you must initialize the driver again with functions 1 and 2.

Function 4	Set speech output parameters
-------------------	------------------------------

Set various parameters for speech output. Among these parameters are the sex of the voice, the pitch, volume and speech tempo.

Input	BX = 4 A = Sex: 0 = male or 1 = female* AH = Register, 0 = low and 1 = high D = Volume 0 thru 9. The default is 5. DH = Pitch 0 thru 9. The default is 5. C = Speech tempo 0 thru 9. The default is 5.	Output	None
--------------	---	---------------	------

* ignores specification of AL = 1 female

Function 5	Speech Output
-------------------	---------------

Start text to speech conversion

Input	BX = 5 ES:DI = FAR pointer to ASCIIZ string containing text to be converted.	Output	AX = 0: OK, text conversion completed 1: Error, specified string is blank 2: Error, string is too long
--------------	---	---------------	--

Sample programs

The CTTS program contains wrapper functions for all the Sound Blaster driver functions. You can add the CTTS.H include file to your programs so they can access the CTTS functions.

Functions from the CTTS module	
Function	Task
ctts_GetDrvVer	Get driver version number
ctts_GetEnvSettings	Return the BLASTER environment variable
ctts_Init	Initialize driver
ctts_SetSpeechParam	Set speech parameters
ctts_Terminate	Uninstall driver
ctts_Say	Convert text to speech

Program VOICE has examples of using the CTTS functions. VOICE is a command line utility for "speaking sentences" through the Sound Blaster card. For example, you could use the following call:

```
voice "this is your Sound Blaster talking"
```

VOICE also accepts the various speech parameters for gender, volume etc. Run VOICE without parameters to display the command's syntax on screen. You'll have to call SBTALKER before starting VOICE. Also make sure the CTTS driver is in the same directory as VOICE.

You'll find the following program(s) on the companion CD-ROM



CTTS.H (C listing)
 CTTS.C (C listing)
 VOICE.C (C listing)
 SBUTIL.PAS (Pascal listing)
 ARGS.PAS (Pascal listing)
 VOICE.PAS (Pascal listing)

38

Processes, Threads And Multitasking

Microsoft is luring developers to their Windows 95 platform by emphasizing features such as the 32-bit model, the OLE interface, the new user interface and the promise of “multitasking.” Arguments over which is the “best system” usually includes these features, especially multitasking. Other operating systems, such as UNIX, have offered multitasking capabilities for a long time. Even OS/2, which was the main competitor to Windows 3.1 at one time, features multitasking capability. However, the multitasking capability of Windows 3.1 is not what many users and developers understand as “true,” or *pre-emptive* multitasking.

Despite all its potential, multitasking by itself cannot solve all problems. You’ll eventually understand why this is true after working with multitasking. Windows 95’s multitasking capability is interesting if you take a quick look at the operating system. Obviously, multitasking is an important advantage in many situations. However, multitasking is not the secret answer for all data problems you’ll face. Many problems cannot be solved with multitasking while other problems are much harder to solve under multitasking. It’s even possible that a program will run slower under multitasking. A certain amount of processor time will be required to forcibly maintain parallel processes. The truth is to use multitasking with careful planning.

Multitasking Under Windows 95

When you ask how Windows 95 loads and starts applications and how the system core manages them while they’re running, you’re likely to hear the terms “process” and “thread.” They refer to the system’s central points regarding running applications. When an application is to start, the operating system first creates a process to which all the application’s resources are assigned. These include the following:

- The address range of the application with the necessary memory management component (e.g., page tables and description tables).
- The physical memory that is assigned to the program for its code, data and stack.
- The command line that was specified with the application call.
- The environment variables used by the application.
- The application’s working directory.
- All resources, such as files and memory areas, that the application acquires by calling API functions while it’s running.

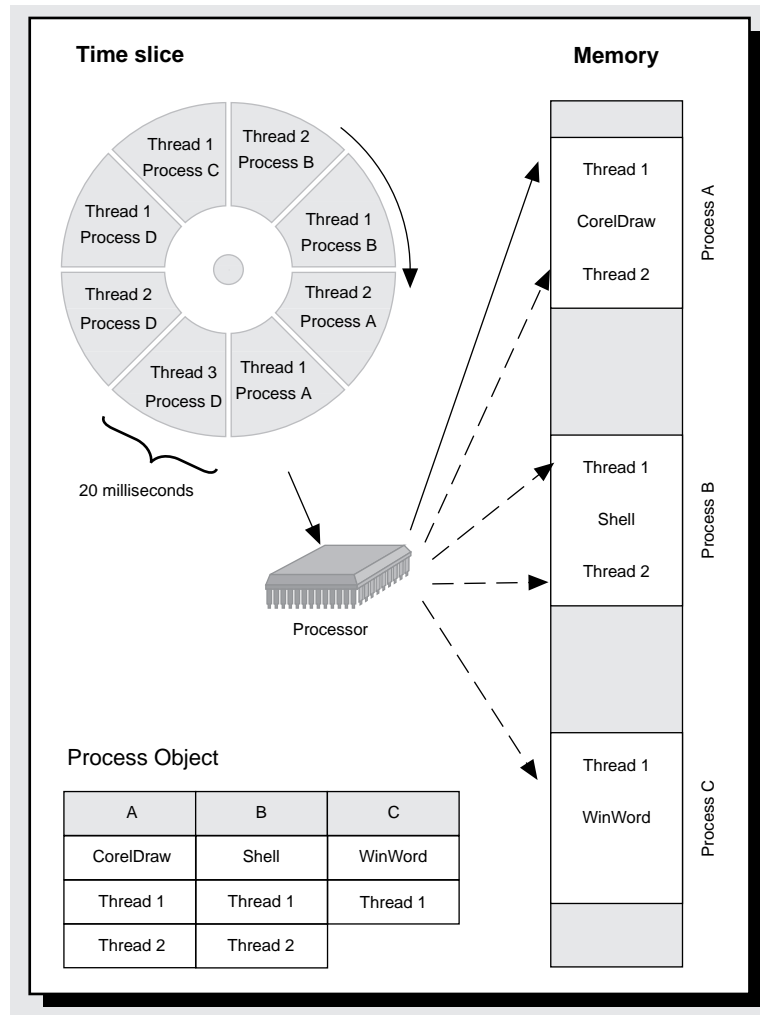
The process object accompanies the application throughout its entire lifespan in memory. The accompanying process object is deleted only when the application is ended and its memory is released. While the application is running, its process acts like an outer shell for the application. It also serves as a construct for the operating system that lets the OS control several parallel applications and shield them from one another. Several active processes also occur when the user has started several applications. One process occurs normally per application.

Threads are located within the individual processes. These are *execution units*. They comprise the process of running the application. The process is not executed when an application is started. Instead, an initial thread called the *primary thread* is started. It’s automatically created by the shell. Therefore, each process has at least one thread that represents the execution of that process.

Multitasking as timeslice processing

Multitasking occurs on the level of threads. In other words, several threads are executed at once under multitasking. However, the apparent parallel execution of these threads is only an illusion. A system with only one processor can only process one piece of program code at any given time. Only powerful multiprocessor systems, currently supported only by Windows NT, can execute more than one thread at any given time because they use multiple processors.

*Multitasking
using the
timeslice
method*



To run several parallel threads using one processor, we need to use the timeslice principle as implemented by Windows 95. The processor in this method is switched between individual threads at short intervals. Thread A is first executed for twenty milliseconds. Then thread B is executed for twenty milliseconds. Finally, thread C is executed for twenty milliseconds. The processor starts over with thread A and repeats this cycle until one of the threads is ended or another is added. Similar to how a movie displays individual pictures in rapid succession to give the illusion of continuous movement, this timeslice method gives the user the impression that the threads are being processed simultaneously.

How soon a thread will get its next turn depends on how fast the processor switches between the individual threads. The faster the processor can make switches improves the appearance of true parallel processing. However, this system has its limits since the operating system needs to perform a certain amount of bookkeeping each time the processor switches from one task to

another. It's necessary, for example, to save the contents of the processor registers each time the processor switches to a new thread. Then they're restored when the processor returns to this thread after giving the others their turns. After all, the execution of a thread may be interrupted at any time and at any point within the program code. The thread will almost certainly crash if the original register settings are not restored when the processor returns to a particular thread.

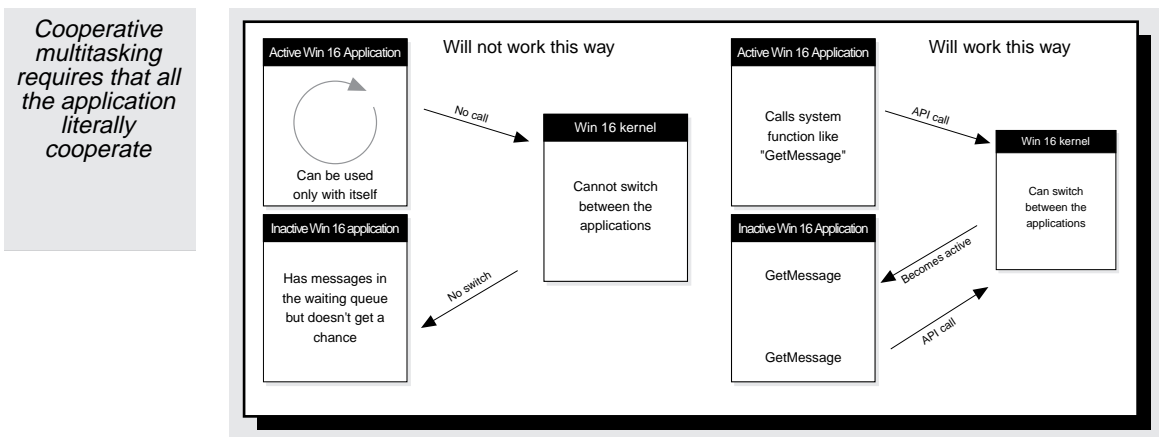
Since the amount of bookkeeping that is necessary with each timeslice is fixed and independent of the duration of the thread's timeslice, these slices must not become too short. Otherwise, the proportion of bookkeeping time to actual processing time will keep increasing. If we were to take this to the extreme, the operating system would spend almost all its time switching between tasks. Little time would remain to process the threads. This is not the desired result.

The timeslice duration in Windows 95 is set to 20 milliseconds. This time corresponds to 50 threads a second. This time is certainly short enough to give the user the impression that the threads are being processed simultaneously. It's also long enough so the processor has enough time within each slice to do some work. If we assume an average command duration of five clock-cycles and a processor clock speed of 100 MHz, the 50 milliseconds of a timeslice correspond to an impressive 400,000 executable machine language commands per timeslice.

Differences between cooperative multitasking

Threads are forcibly interrupted and paused with the timeslice method called *pre-emptive multitasking*. This is different from *cooperative multitasking* used by Windows 3.1 (and still used by Windows 95 for 16-bit applications). To switch the processor from one application to the next, the Windows 3.1 kernel must monitor the application's activity. It can only switch to the next thread when an application makes an API call such as `GetMessage()` or `PeekMessage()`. These API calls allow the operating system to regain control over the processor. The kernel uses this opportunity to continue executing another application that had also called one of these two functions and was thus blocked by the system kernel.

Applications under Windows 3.1 therefore must quickly evaluate any messages they receive after making a `GetMessage()` or `PeekMessage()` call. This must be done so they can then immediately call one of these two functions again. The speed at which this occurs determines how quickly the processor can execute another application. However, if an application performs another, longer task after receiving a message, the cooperative system of multitasking begins to stall, that is, if the application doesn't make a `GetMessage()` or `PeekMessage()` call at some point soon. This is the only way the system kernel can move on and execute another application that is waiting, before returning to the application that was interrupted.



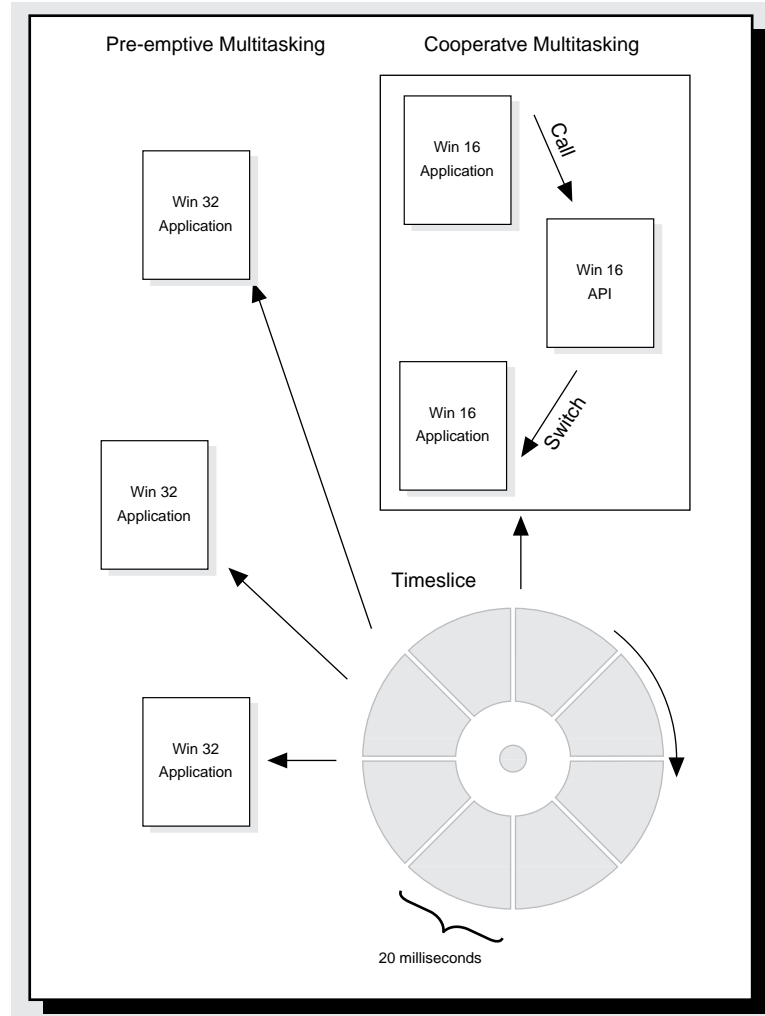
Cooperative and pre-emptive multitasking side-by-side

Windows 95 must support both 32-bit applications and the earlier 16-bit applications. Therefore, it must support both Win32's pre-emptive multitasking and Win16's cooperative multitasking. All Win16 applications are therefore executed in a so-called virtual machine (VM), which simulates the normal 16-bit Windows environment. Only cooperative multitasking is used within this VM. In other words, the operating system only switches from one 16-bit application to the next when the current

application makes a GetMessage(), PeekMessage() or Yield() call. The Win16 VM, however, shares the processor with all the other active Win32 applications through pre-emptive multitasking.

The active Win16 application is therefore only processed when the VM within which it is running receives a timeslice. Within that timeslice, the VM can then use cooperative multitasking to switch to the next Win16 application. When the VM loses its timeslice, the processor moves on to the next Win32 application. In this manner, the Win16 applications are switched using cooperative multitasking, like before, without compromising the pre-emptive multitasking used for the Win32 applications.

*Using
pre-emptive and
cooperative
multitasking
side-by-side*



Multiple threads in one process

Multitasking as implemented in Windows 95 doesn't end with just one thread per process. Its most important ability is to use multiple threads in a process. An application can therefore set up a parallel execution of multiple tasks within itself. This means that multitasking occurs not only between individual applications but also between the individual threads of an application or process. This allows an application to initiate and conduct several tasks at once (parallel to one another).

Threads share the resources allocated to their process, thereby collectively accessing the global variables and sharing the operating resources that were acquired by the process. These include memory, files, communication channels, etc. However, a thread cannot access the operating resources of a thread in another process. Processes and their threads are strictly isolated

from others. Even the threads within a single process are shielded from one another in that each thread receives its stack on which its local variables are stored.

Like any application, a multithreaded process starts with only a single thread. It then uses API calls during its runtime to create further threads. However, this approach is not always an advantage. You'll want to decide on a case by case basis if an application can benefit from being split into several threads. The following lists examples of how you can use multiple threads:

- **Fore and background threads:** An application is split into a foreground thread and a background thread. The foreground thread is responsible for the interaction with the user and for controlling the interface. The background thread processes the data that was entered. It then prints the data, saves the data to disk or uses the data for calculations. By doing this, it remains responsive to the user's input although the application is actively working on the data.
- **Client / server applications:** A server is to answer requests by several clients simultaneously. Therefore, a separate thread is created for each request received by a client. Each of these threads is responsible for responding to the request and is deleted when it has answered the client. The number of threads thus changes with the number of requests that are being processed simultaneously.
- **Simulations / games:** Threads can be useful when many objects need to move across the screen and through space. Each object is represented by a thread, thereby receiving a certain amount of processor time with each simulation cycle. This allows the object to move and interact with elements in its environment.

These three scenarios are different from one another regarding their necessary code structures and considerations in using multitasking. We'll use many similar examples throughout this chapter.

Threads are of little use when there is no need to perform tasks on a parallel basis, where no complex tasks need to be performed while the system remains open to user input. An example is the classic dialog applications that receive data from the user in a screen mask, quickly store the data and then restart it. Before dividing your program into several threads, consider if this is an advantage. After all, using multiple threads means more work since the individual threads must be managed relative to one another. This management requires additional program code.

Priorities and scheduling

If several threads are waiting to be active at any one time, the operating system must decide which thread will receive how much processor time at which instant. Is it best to treat them all equally or should certain threads receive priority?

If we look at the example from above, which uses a foreground and a background thread, the answer becomes fairly clear. The foreground thread has the task to respond promptly to the user's input. If the background thread is active when the user presses a button, the foreground thread must be processed immediately. This is true whether the background thread has completed its timeslice or is still in the middle of being processed. This means the threads must be assigned different priority levels according to their importance. The background thread must wait whenever the foreground thread needs to become active. The scheduler determines which thread is processed and which has to wait. So, the scheduler is one of the most important components in the Windows 95 system since it assigns processor timeslices to the individual threads.

However, a mere distinction between foreground and background threads is insufficient for the demands of true multitasking. First, a more detailed comparison is necessary since several classes of threads with different requirements regarding their processing priority are possible. Furthermore, the priority model must be dynamic rather than static if the system is to respond to user interaction. A thread that previously had a lower priority will often require a higher priority level because the user activated the window of that application so it's now the active application. This is called a *priority boost*.

Priority classes

The difference between priority levels in Windows 95 is based on four priority classes. A process will be assigned one of these four classes, and the threads of this process will, at first, also receive this priority. The priority classes are: Realtime, High, Normal and Idle. The following table explains each priority class:

Class	Task
REALTIME_PRIORITY_TASK	This class is not for application programs. It's instead for device drivers and other components that must react immediately to hardware events.
HIGH_PRIORITY_CLASS	Should be assigned to "hotkey" applications, i.e., programs that wait in the background for a specific event, such as the pressing of a specific hotkey, and then react immediately. Windows 95 Task Manager (a+t) is an example of such an application.
NORMAL_PRIORITY_CLASS	Intended for all normal applications that have no special scheduling.
IDLE_PRIORITY_CLASS	Threads of this class are only processed when no threads of a higher class are ready to be processed. This class is only used for threads that perform housekeeping tasks in the background (i.e., garbage collection or reserving or clearing memory ahead of time).

You might consider assigning the Realtime priority class to your process so it's given as much processor time as possible. However, this will not make an application faster. The internal system threads responsible for handling mouse and keyboard events run exclusively under High priority. So, unless the Realtime thread is blocked for some other reason, it's possible the system will be unable to respond to mouse and keyboard input promptly since its threads have a lower priority class.

However, most processes are set at normal priority. The following is an example of a process working perfectly using the normal priority class: An application that receives user input through menus and dialog boxes, processes them within the program, and then generates some sort of output. In the following section, on controlling processes using the Windows API, we'll discuss how to set the priorities of a process.

Thread priorities

The threads of a process are assigned priority levels within the priority class of the process (you can consider it fine-tuning their priorities). Five different priority sublevels are used for this purpose: Lowest, Below Normal, Normal, Above Normal and Highest. All threads begin in the Normal subclass but they can change this setting using the appropriate API functions.

However, the scheduler doesn't only make its decision on how to divide processor time based on statically determined thread priorities. It instead uses a dynamic priority that it manages internally for each thread. Each thread is first initialized with the static priority, which is then adjusted dynamically based on current system requirements. While a thread doesn't leave the priority class of its process, it may be bumped up from Normal to High. An example of this is when a system event has occurred to which it must react quickly. The thread is then stepped back down by one subclass with each further timeslice that it receives. This continues until the thread is returned to its original priority.

Scheduling in detail

The four priority classes of each process with their five subclasses are represented within the system kernel in a priority table with values from 0 to 31. Higher priority values represent a quicker time for a thread to receive its next timeslice from the scheduler. First, the priority classes of the processes are weighed according to the table on the right.

Depending on the thread priority (subclass), the base priority value is increased or decreased by the values in the table on the lower right. Adding the values in the two tables doesn't result in the minimum value 0 nor the maximum value 31. These priority values are only used for system internal threads that remain hidden to the user.

Priority class	Base priority
Idle	4
Normal	9 if the process is the current window 7 if the process is not the current window
High	13
Realtime	24

Thread priority	Priority increment	Thread priority	Priority increment
Lowest	-2	Above Normal	+1
Below Normal	-1	Highest	+2
Normal	+0		

The scheduler is surprisingly harsh in assigning processor time. Only the threads with the highest priority receive a timeslice. If there are three threads with a priority level of 16 and none with a higher priority, it will process these threads alternately without assigning a timeslice to any other thread with a lower priority. This may seem contradictory to the whole idea since the internal system threads for monitoring the mouse and keyboard already are at a higher priority than normal applications. How are applications supposed to get any processor time? The answer is quite simple: the mouse and keyboard threads are blocked from scheduling when they have nothing to do. The scheduler simply ignores them if no mouse or keyboard events have occurred. This allows threads with lower priorities to be processed.

So how does the system know that a thread is simply waiting and therefore doesn't need to be scheduled? This can happen in several ways. First, threads use different API functions to notify the system that they are waiting for a specific event. These events include pressing a key, completing a hard drive access or ending another process. Threads can also intentionally "pause" themselves for a specific time before they're processed again. This is also accomplished using API functions. Lastly, threads can also be prevented from being processed further when they make a GetMessage call. If the message queue doesn't contain a message for this thread, it is paused so the processor can continue with other threads of the same or lower priority.

When the system places a message in the queue of this paused thread, it is released for further processing. If another thread with a lower priority is active in this instant, it is interrupted immediately so the paused thread can be continued. While this thread waited for the answer to its GetMessage() call, the processor was able to work on other threads. However, the former will also be interrupted immediately if another paused thread with a higher priority level is suddenly released (perhaps a system thread that was waiting for a hard drive operation to be completed). This ensures the following:

- The threads with the highest priority are processed when possible.
- Threads with comparatively lower priority levels are processed when the processor is not needed by a higher level thread.

Although this may sound complicated, most programmers normally are not concerned with these details. However, it's apparent the system kernel has quite a task in distributing processor time efficiently and making certain the system always responds promptly.

Processes

The Win32 API under Windows 95 provides many different functions that you can use to control processes. These include functions for starting and terminating processes, accessing a foreign process and functions for setting the priority level and checking individual process properties. These properties can include the exit code, the current priority level or the initial window in which a process was started.

Win32 functions for controlling processes	
Function	Task
CreateProcess	Creates a process.
OpenProcess	Returns a handle to an existing process.
ExitProcess	Ends the current process and all of its threads.
TerminateProcess	Terminates a specific process from the outside.
SetPriorityClass	Sets the priority class of a specific process.
GetCurrentProcess	Returns a pseudo-handle for the current process.
GetCurrentProcessId	Returns the process ID of the current process.
GetExitCodeProcess	Returns the exit code of a specific process.
GetPriorityClass	Returns the priority class of a specific process.
GetStartupInfo	Returns information from the STARTUPINFO data structure with which the current process was started.
WaitForInputIdle	Waits until a specific process is idle.

Starting a process

The life of a process starts with the API function call `CreateProcess()`. The Windows Shell or an alternative shell will normally make this call to start an application upon the user's request. This function replaces the 16-bit functions `WinExec()` and `LoadModule()`. Although these are still supported, they simply trigger a `CreateProcess()` call internally. Any application, and not just the shell, can use `CreateProcess()` to create a process that is then executed parallel to all the other running processes. Of the many functions for controlling processes, `CreateProcess()` is the one with the most parameters and options.

```

BOOL CreateProcess(
    LPCSTR          lpApplicationName, //pointer to the name of the EXE file
    LPSTR           lpCommandLine,     //pointer to command line
    LPSECURITY_ATTRIBUTES lpProcessAttributes, //Win95: NULL
                                           lpThreadAttributes, //Win95: NULL
    BOOL            bInheritHandles,    //inherit Handles?
    DWORD           dwCreationFlags,    //priority and process type
    LPVOID          lpEnvironment,      //pointer to environment variables
    LPCSTR          lpCurrentDirectory, //current directory
    LPSTARTUPINFO    lpStartupInfo,     //Startupinfo
    LPPROCESS_INFORMATION lpProcessInformation //read info through new P.
);

```

lpApplicationName, lpCommandLine

The function's first argument is a pointer to a C string containing the name of the file that is to be executed, if necessary with the correct path and drive letter. If NULL is specified, the application name must be found as the first entry in the command line string specified by the `lpCommandLine` parameter. Otherwise, the string referenced here is used to pass any necessary information on the application that is to be started. Like under DOS, this allows an application to obtain information from its command-line startup-call. This string receives the command line arguments either through the API function `GetCommandLine()` or by accessing the arguments `argc`, (or `argv`) if it is a console application written in C.

lpProcessAttributes, lpThreadAttributes

These parameters can be ignored for Windows 95 programming; specify NULL for these parameters. They're used by the security system of Windows NT (not included with Windows 95).

Note concerning security attributes under Windows 95

The security attributes that are used by several process and thread management API functions derive from the NT version of the Win32 interface. They're unnecessary under Windows 95 since the extensive security architecture implemented in NT was not incorporated into Windows 95. This security architecture under NT is used to authorize all user actions. Each API call must ultimately be attributed to a specific user. The system then authorizes or denies the API function call depending on the settings of the particular user account. This allows the file related functions to use the security attributes to determine, for example, whether a particular user may change the files in a particular directory.

However, since this constant validation of access rights requires large processor time and memory, it was not integrated into Windows 95. Although Windows 95 does support some security functions (for example, network access), it does not validate each API function call.

To make porting between Windows 95 and NT easy, these security parameters with functions such as `CreateProcess()` are also present under Windows 95. Simply enter NULL for these parameters under Windows 95 instead of specifying a pointer to some security structure. This will also make it easier to port applications to NT since with these security parameters NULL represents the security attributes of the current user.

bInheritHandles

This flag determines whether handles of the parent process (the process that places the `CreateProcess()` call) are inherited by the new process. In this case (`bInheritHandles = TRUE`) these handles permit new process gains access to objects belonging to the parent process, such as a file that was opened by the parent or a communication port. However, to make this possible the parent process must first communicate the value of the handle to its child process. This value can be communicated through an appropriate parameter in the command line, for example, or through the environment variables. The parent process and

the child process must thus work together to be able to use the same resources together. It's not enough to simply inherit them. However, not all objects that are referenced are available to the child process. The following table shows the distribution:

Handles can be inherited by	Handles cannot be inherited by
Threads and processes	Allocated memory
Files	GDI objects (brushes, pens, etc.)
Memory mapped files	USER objects (windows, carets, etc.)
Mailslots	Synchronization objects (events, mutexes, semaphores)
Communication ports	The console of the parent process, if it is a console application

dwCreationFlags

This parameter determines the type and priority of the new process. A series of flags are defined that you can combine. Some flags are reserved for debuggers, console applications or special execution modes for 16-bit Windows applications. They're not needed for starting normal Windows 95 or DOS applications. The most interesting flag, particularly regarding multitasking and the cooperation of two processes, is `CREATE_SUSPENDED`. It makes certain the new process is first paused. This allows the caller to start the process later, which can be done by simply calling the `ResumeThread()` API function.

Flag	Description
<code>CREATE_SUSPENDED</code>	Execution of the initial thread in the new process pauses until a <code>ResumeThread()</code> call is made.

Unless the new class is to automatically be assigned the Normal priority class, you also must specify one of the following constants in the `dwCreationFlags` parameter:

Priority flag	Priority class
<code>REALTIME_PRIORITY_CLASS</code>	Only for hardware
<code>HIGH_PRIORITY_CLASS</code>	Respond to input very quickly (hotkey applications)
<code>NORMAL_PRIORITY_CLASS</code>	Normal applications
<code>IDLE_PRIORITY_CLASS</code>	Execute only when system is not busy

lpEnvironment

Each process always has a current directory and a set of environment variables. The environment variables are determined by the `lpEnvironment` parameter when the `CreateProcess()` function call is made. If you also specify `NULL` for this parameter, the new process will receive the same environment variables as its parent process. If the child process is to receive different environment variables, the parent will have to construct a new block using the environment variables and specify a pointer to this block in `lpEnvironment`.

Environment variables

Environment variables are a simple way to supply an application with information from the outside, such as the name and path of a file, or perhaps a password. A familiar environment variable, from DOS is `PATH`. It specifies the search path for files. Environment variables normally follow this format:

```
name=value
```

They're structured as C strings (with a terminating NUL byte) and are grouped in memory into one block of environment variables. Each environment variable from the block's start to its end borders directly on the preceding and following ones. The start of each new environment variable is thus denoted by the terminal NUL byte of the preceding variable. The end of the block is marked by an empty string. In other words, the NUL byte of the last environment variable is followed by another NUL byte. This ends the block.

```
"PATH=C:\\WINDOWS\\0",
"DATA=Monthly_Anaylsis\\0",
"\\0"
```

After `CreateProcess()` has been successfully executed, the parent can discard the contents of the block with the environment variables that it created. This is possible because it sets up its own memory block containing the environment variables for the child process.

lpCurrentDirectory

Another pointer to a string is required for the `lpCurrentDirectory` parameter. This string represents the working directory of the applications that are to be started. When these access directories and files with read or write operations without explicitly specifying a path, this working directory will be used as the starting point. If `NULL` is specified for this parameter, the application that is being started will receive the same working directory as its caller.

The current drive is always determined by the current directory. However, depending on the application, this may not be enough. If an application is working with several drives, it will need a separate directory for each drive. The Windows 95 I/O subsystem supports the use of multiple current directories using special environment variables. An application can pass these variables to its offspring processes. Each of these environment variables determines the current directory for one drive and must follow this format:

```
=[LW:]=[path with drive letter and current directory].
```

You could use the following environment variables to specify, for example, the default directories for drives D: and F:

```
=D:=D:\DATA\CALCULATIONS
=F:=F:\BIN2\BUILD363\DEBUG
```

These environment variables are needed whenever the application calls one of the file functions from the Win32 API and only specifies the drive letter, such as "F:". In this case, the I/O subsystem will search the application's environment variables for an entry that conforms to "=F:=" and then will then use the specified directory to perform the function. If such an entry cannot be found, the drive's root directory is used as the default directory.

lpStartupInfo

This parameter is used to influence the display of the window in the new process. The parameter must point to a structure of the `STARTUPINFO` type. Specify `NULL` if you don't want to use the options that can be selected using `STARTUPINFO`.

```
typedef struct _STARTUPINFO {
    DWORD    cb;                // must be initialized with sizeof(STARTUPINFO)
    LPSTR     lpReserved;       // NULL
    LPSTR     lpDesktop;        // NULL
    LPSTR     lpTitle;          //only for consoles: window title
    DWORD     dwX;              //X ordinate, upper left-hand corner of window
    DWORD     dwY;              // ditto, Y abscissa
    DWORD     dwXSize;          // width of the window
    DWORD     dwYSize;          // height of the window
    DWORD     dwXCountChars;    // only for consoles: number of columns
    DWORD     dwYCountChars;    // only for consoles: number of rows
    DWORD     dwFillAttribute;   // only for consoles: background color
    DWORD     dwFlags;           // accept which fields?
    WORD      wShowWindow;
    WORD      cbReserved2;      // 0
    LPBYTE     lpReserved2;     // NULL
    HANDLE     hStdInput;       // only for consoles: handle of default input device
    HANDLE     hStdOutput;      // only for consoles: handle of default output device
    HANDLE     hStdError;       // only for consoles: handle of default error device
} STARTUPINFO;
```

Most of the fields in STARTUPINFO are specific to console applications. For more information, see the Chapter/Section on consoles. The most interesting fields for normal applications, however, are dwX, dwY, dwXSize and dwYSize. These make it possible to determine the size and location of the main window of the new process. That is, if the new process specifies the CW_USEDEFAULT constant for the window position, size or both with the CreateWindow() function call that creates the window. This way, the predefined settings are used.

The window display mode (maximized, minimized, normal, etc.) can also be specified. Again, the new process must cooperate for these settings to be used. It receives the flag specified in STARTUPINFO.wShowWindow as the last parameter (nCmdShow) associated with its WinMain() start function. This will allow the parent process to determine the window display mode, provided the new process specifies this flag with its CreateWindow() call. The parent process has to specify one of the familiar SW_...constants in STARTUPINFO.wShowWindow, like the ones required by ShowWindow(), such as SW_SHOWMAXIMIZED (full screen) or SW_SHOWMINIMIZED (icon).

If you want to use STARTUPINFO to specify one of the aforementioned parameters, you'll need to load the dwFlags field with one of several STARTF_...constants. They tell CreateProcess() which fields of STARTUPINFO contain information that is to be passed to the child process that is being created.

Flag	Description
STARTF_USESHOWWINDOW	Use the contents of wShowWindow
STARTF_USESIZE	Use the contents of dwXSize and dwYSize
STARTF_USEPOSITION	Use the contents of dwX and dwY
STARTF_FORCEONFEEDBACK	Display StartGlass cursor
STARTF_FORCEOFFFEEDBACK	Don't display StartGlass cursor

The last two flags in this table concern a new Windows 95 cursor called the StartGlass cursor. It can be displayed while an application is being launched. This cursor is meant to inform the user that he or she can work with other applications while the new one is being started. This cursor, therefore, combines the arrow with the hourglass.



Display this cursor by specifying STARTF_FORCEONFEEDBACK in STARTUPINFO.dwFlags. The system will then watch the activities of the launched application and switch the cursor back to its normal arrow appearance if the application fails to make a GDI call within the first two seconds after being started. Otherwise, the StartGlass cursor will remain until the application makes its first GetMessage() call. Only then does the application window's default cursor appear. If you want to suppress the display of the StartGlass cursor, simply specify the constant STARTF_FORCEOFFFEEDBACK in STARTUPINFO.dwFlags.

lpProcessInformation

The caller doesn't receive information on the newly created process through the return value of CreateProcess(), but rather through a data structure of the PROCESS_INFORMATION type, which the process must provide for this purpose. CreateProcess() expects a pointer to this structure in the lpProcessInformation parameter and returns the following information in the structure after the successful completion of the function:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE   hProcess;           // Handle of the newly created process
    HANDLE   hThread;           // Handle of the initial thread
    DWORD    dwProcessId;       // ID of the newly created process
    DWORD    dwThreadId;        // ID of the initial thread
} PROCESS_INFORMATION;
```

The reason two references (as handles and IDs) to the process and its initial thread are used is twofold. Since the ID is valid across the entire system, it separates the process or its thread from all other threads and processes in the system. Only one process or thread with a given ID is available at any one time. All functions that facilitate the communication between processes therefore use process IDs as references to the processes in question.

The vast majority of API functions related to processes and threads, on the other hand, use process and thread handles. Handles are always only locally valid, within each process. They cannot be used apart from their processes and are meaningless for other processes and their threads. Handles are also not unique. Therefore, two processes may have the same handle for entirely different objects. The system prefers handles, because it doesn't need to coordinate them globally, but can simply resort to a separate pool of handles for each process.

Return-value

If the desired process was created successfully, `CreateProcess()` will return the value `TRUE`. If `FALSE` is returned, use the `GetLastError()` function to receive additional error information. The following example uses a `CreateProcess()` call. The `STARTUPINFO` structure is used to determine the size, position and display mode of the new process' main window.

```
STARTUPINFO      StartInfo;
PROCESS_INFORMATION ProcInfo;

//— first set Startup Info —————

StartInfo.cb      = sizeof( STARTUPINFO );
StartInfo.lpReserved = NULL;
StartInfo.lpDesktop = NULL;
StartInfo.lpTitle   = NULL;                // only for consoles
StartInfo.dwX       = 100;                 // setting for
StartInfo.dwY       = 100;                 // upper left-hand corner of window
StartInfo.dwXSize    = 300;                // setting for width
StartInfo.dwYSize    = 300;                // and height
StartInfo.dwXCountChars = 0;               // only for consoles
StartInfo.dwYCountChars = 0;               // only for consoles
StartInfo.dwFillAttribute = 0;             // only for consoles
StartInfo.dwFlags     = STARTF_USESHOWWINDOW | // which settings
                      STARTF_USEPOSITION |    // shall be
                      STARTF_USESIZE;         // implemented?
StartInfo.wShowWindow = SW_SHOWMAXIMIZED;    // display mode setting
StartInfo.cbReserved2 = 0;
StartInfo.lpReserved2 = NULL;
StartInfo.hStdInput    = NULL;              // only for consoles
StartInfo.hStdOutput   = NULL;              // only for consoles
StartInfo.hStdError    = NULL;              // only for consoles

//— start process and display info through returned
if( !CreateProcess( "APPLICAT.EXE",          // lpApplicationName
                  "CommandLineParams",      // lpCommandLine
                  NULL,                      // lpProcessAttributes
                  NULL,                      // lpThreadAttributes
                  FALSE,                     // bInheritHandles
                  0L,                        // dwCreationFlags
                  NULL,                      // lpEnvironment
                  "C:\\TEMP",                // lpCurrentDirectory
                  &StartInfo,                // lpStartupInfo
                  &ProcInfo ) )             // lpProcessInformation
    MessageBox( NULL, "Process cannot be created!", "Error", 0 );
else
{ // display process and thread
    printf( "hProcess: %ld, hThread: %ld\n",
           ProcInfo.hProcess,
```

```

        ProcInfo.hThread );

printf( "dwProcessId: %ld, dwThreadId: %ld\n",
        ProcInfo.dwProcessId,
        ProcInfo.dwThreadId );
}

```

However, the settings in STARTUPINFO only take effect because the new process specifies the constant CW_USEDEFAULT for the corresponding parameters when it creates its startup window using CreateWindow(). This is shown in the following partial code:

```

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpzCmdLine,
                   int nCmdShow )
{
    HWND hMainWnd;
    MSG msg;

    //— create main window —————
    hMainWnd = CreateWindow( "CLASS",
                           "CAPTION",
                           WS_OVERLAPPEDWINDOW,           // Style
                           CW_USEDEFAULT,                  // Use settings
                           CW_USEDEFAULT,                  // StartUpInfo
                           CW_USEDEFAULT,                  //
                           CW_USEDEFAULT,                  //
                           NULL,                             // hparent
                           NULL,                             // hmenu
                           NULL,                             // hinst
                           NULL );                          // lpvParam

    ShowWindow( hMainWnd, nCmdShow );                      // display window

    //
    //
    //

```

The start function of a process

If CreateProcess() is used to start a Windows application, not a console application, the execution of the process will start with the WinMain() function, as usual. However, this function call is not made directly by the kernel since it first starts Windows applications with the WinMainCRTStartup() function from the runtime library of the C compiler. WinMainCRTStartup() initializes the global variables and all necessary data structures of the runtime library, for example for the heap provided by malloc(). It also evaluates the command line and obtains a pointer to the environment variables of the new process. Once all these tasks have been completed, the WinMain() function of the application is called and the control of the program execution is thereby passed to the new process. WinMain() receives the following parameters when it is called:

```

int WINAPI WinMain(
    HINSTANCE hInstance,           // instance handle of the process
    HINSTANCE hPrevInstance,      // instance handle of the preceding process
    LPSTR lpCmdLine,              // pointer to the command line
    int nShowCmd                  // flag for the display of the main window
);

```

hInstance

Handle to the instance. Many API calls require this to identify the process. This handle under Win32 is identical to the module handle, which is required by several functions. You can, therefore, specify the value in *hInstance* wherever the module handle of the current process is required.

hPrevInstance

Points to an instance of the process under Win16 that may have already started. Many applications use this parameter to access the preceding instance. This is no longer possible under Win32 since processes are executed in separate address ranges and so do not have access to other processes. Therefore, the value NULL is generally specified for this parameter under Win32.

lpCmdLine

This pointer points to a C string containing the command line with which the process was started.

nShowCmd

This is the recommended flag for the *nCmdShow* parameter in *ShowWindow()*. It contains one of the familiar SW_... constants, such as SW_HIDE, SW_MINIMIZE, etc. Processes should use this flag when displaying their main windows, so the parent process is able to determine the initial display mode of the new application window.

The predefined variables of the runtime library

During its initialization, the runtime library loads a whole series of global variables through which a process can obtain information on the current operating system version, the arguments from the command line, and its environment variables.

unsigned int _winmajor	The main-version number of the operating system under which a process is running. For Windows 95, this value is 4.
unsigned int _winminor	The subversion number of the operating system, consisting of two digits. For the second version of Windows 95, for example, this value would be 20..
unsigned int _winver	Combines winmajor and winminor. winmajor is in the Hi byte, winminor in the Lo byte.
unsigned int _osver	The operating system's current build-number. The last build put out by Microsoft just before its ultimate publication of Windows 95, for example, bore the number 950.
unsigned int __argc	The number of arguments from the command line, the way a pure C program receives it through the argc parameter of the main() function when it is started.
char ** __argv	Corresponding to argc, a pointer to an array containing pointers to the individual parameters from the command line, as strings.
char ** _environ	A pointer to an array containing pointer, each of which points to one of the process' environment variables.

Accessing the environment variables

The Win32 API has four functions to manage the environment variables of a process. These functions let you set and read individual environment variables and use environment variables to manipulate memory blocks.

Function	Description
FreeEnvironmentStrings	Frees a data block with environment variables
GetEnvironmentStrings	Provides a pointer to the data block with the environment variables of the current process
GetEnvironmentVariable	Reads the value of an environment variable of the current process
SetEnvironmentVariable	Determines the value of an environment variable of the current process.

If a process needs to manipulate its environment variables “by hand” or make a copy of the block with environment variables, it can obtain a pointer to its current block containing environment variables using *GetEnvironmentStrings()*.

```
LPVOID GetEnvironmentStrings(VOID);
```


It's possible by using the pointer that is returned to determine the number of available environment variables, their length and the total length of the block. However, the application must run through the block with the environment variables on its own since the WIN32 API does not provide a function that would perform this task for the application. The structure of such a block (simply a sequence of C strings) makes it easy for the application to obtain the necessary information on its own.

When it's necessary to read or set specific environment variables, however, use the API functions `GetEnvironmentVariable()` and `SetEnvironmentVariable()`.

```
DWORD GetEnvironmentVariable(
    LPCTSTR lpName,          // pointer to a string containing the name of the
                              // environment variable
    LPTSTR  lpBuffer,        // pointer to the buffer that is to receive the
                              // value of the environment variable
    DWORD   nSize            // size of the buffer for the value of the environment
                              // variable
);
```

When it's called, `GetEnvironmentVariable()` expects a string containing the name of the desired environment variable. Furthermore, a pointer to a buffer is needed in which the function can store the value of the desired environment variable. Also, the caller must specify the buffer's length. Then the function knows the value will fit into the buffer that is provided.

The function result returned by `GetEnvironmentVariable()` is the length of the value (minus the string's terminating NUL-byte). The desired environment variable was not found if the value 0 is returned. If the return value is greater than the length of the buffer specified with the function call, the value of the environment variable could not be copied into the specified buffer, since it was longer than the available buffer. In this case, the return value represents the required buffer length, including room for the terminating NUL-byte.

Use the `SetEnvironmentVariable()` API function to set, rather than read, environment variables. With its function call, it expects two pointers to C strings. The first pointer is the environment variable's name. The second pointer is its new value. The specified environment variable will be created if it does not exist. If the variable already exists, it will be assigned the new value.

```
BOOL SetEnvironmentVariable(
    LPCTSTR lpszName,        // pointer to a string containing the name
                              // of the environment variable
    LPCTSTR lpszValue        // pointer to a string containing the new
                              // value for the environment variable
);
```

This function also makes it possible to delete an environment variable. Simply specify NULL for the pointer to the string with the variable's new value. Lastly is the `FreeEnvironmentStrings()` API function. It's used to drop the entire block with environment variables. The function requires a pointer to the block, which can be obtained using `GetEnvironmentStrings()`. After `FreeEnvironmentStrings()` has been executed successfully, the current process has no more environment variables.

```
BOOL FreeEnvironmentStrings(
    LPTSTR lpszEnvironmentBlock // pointer to the block of environment
                              // variables to be freed
);
```

The two code sequences are intended to show how environment variables are used. The first sequence determines the length of the block of environment variables. The Win32 API offers no special function for this purpose. However, with a little effort, we can make it through the block and determine its length.

```
LPSTR lpEnviron;          // pointer to the block of environment variables
int    iLen;              // running counter for the length
```

```

iLen = 0; // initialize length
lpEnviron = GetEnvironmentStrings(); // get pointer to the block of
//environment variables

if( lpEnviron ) // is there such a block?
while( *lpEnviron ) // is there another environment variable?
{ // yes, pointer does not point to an empty
    int iSize; // for the length of the current environment variable

    iSize = strlen( lpEnviron ); //obtain length of the current environment
//variable
    iLen += iSize; // add to total

    // set pointer to next string (strlen() does not include \0)
    lpEnviron += iSize + 1;
}

printf("Length of the environment block: %ld ", iLen );
// display length of block

```

The second sequence shows a process that starts a new process. However, it's the first to remove an environment variable with the user's password from the environment block. After all, it's better not to propagate sensitive information like passwords too freely. Finally, the password must be restored to the environment variables of the parent process.

```

char szPassword[ 50 ]; // reads the password

GetEnvironmentVariable( "Password", // temporarily save the current password
    szPassword,
    sizeof( szPassword ) );

// delete the password environment variable,
// so it's not inherited by the new process
SetEnvironmentVariable( "Password", NULL );

CreateProcess(...); // create new process

// restore password
SetEnvironmentVariable( "Password", szPassword );

```

Ending a process with a misxpel

If a process is not ended by the end of the WinMain() function, use two functions from the Win32 API instead. The system prefers using ExitProcess() to end the current process. The application's exit code can be specified through the parameter uExitCode. ExitProcess() does not return to the caller, but simply ends the current process directly. All currently running threads of the process are terminated immediately. Its offspring processes the process has started during its runtime, however, continue to run independently of the terminated process.

The ExitProcess() call notifies all the DLLs that the process or its threads has accessed of the termination of the process. This allows the DLLs to free all the resources that they have allocated to the process as their "user".

Function	Description
ExitProcess	Ends the current process and all of its threads
TerminateProcess	Ends a specific process.
GetExitCodeProcess	Returns the exit code of a specific process.

```

VOID ExitProcess(
    UINT uExitCode           // exit code of the process
);

```

The system will automatically close any resources that the process may leave open after its termination. However, the process itself should also free all the resources (memory, files, etc.) that it may have acquired before `ExitProcess()` is called. `ExitProcess()` is automatically called by a C program when it ends the `WinMain()` function. The call is then made in the startup code of the C runtime library. The return value of `WinMain()` is passed to `ExitProcess()` as the exit code of the process. It's also possible for any thread of a program to call `ExitProcess()` at any desired point to end the application.

While `ExitProcess()` can only be used by a process to terminate itself, `TerminateProcess()` makes it possible to terminate other processes. The system uses this function, for example, after an Exception is triggered to end the process without its help. However, use `TerminateProcess()` only in emergencies. Otherwise, use `ExitProcess()` whenever possible. `TerminateProcess()` does not inform the DLLs of the process of its termination so they cannot end their tasks properly. Although resources allocated for the process within the DLL are freed automatically when the process is terminated, data that might still be in the DLL's internal buffers which still hasn't been written to disk, for example, will be lost.

```

BOOL TerminateProcess(
    HANDLE hProcess,           // process handle
    UINT uExitCode           // exit code of the process
);

```

A process can obtain the exit code of another process using `GetExitCodeProcess()`. However, to do so, the process must have a handle of the process that it wants to close. The handle must be passed to the function as the first parameter. The second parameter is a pointer to a `DWORD` or `LONG` variable that is to receive the exit code of the specified process. If the exit code was obtained successfully, `GetExitCodeProcess()` returns the value `TRUE`. If the specified process is still running, the designated `LONG` variable will contain the value of the constant `STILL_ACTIVE` after the function call has been made.

```

BOOL SetPriorityClass(
    HANDLE hProcess,           // process handle
    DWORD fdwPriority          // priority class flag
);

```

Setting and reading priorities

Although the priority class of a process and its initial thread must be set with `CreateProcess()` when the process is created, it's still possible to change it during runtime using `SetPriorityClass()`. It's also possible to ascertain the priority call of a process at any time using `GetPriorityClass()`. All that's needed for reading the priority class is the process handle. It must be passed to `GetPriorityClass()`. If a process wants to determine its own priority class, it must first obtain a handle to itself using `GetCurrentProcess()`.

Function	Description
<code>SetPriorityClass</code>	Sets the priority class of a given process
<code>GetPriorityClass</code>	Returns the priority class of a given process

```

DWORD GetPriorityClass(
    HANDLE hProcess           // handle of the process
);

```

The return value of `GetPriorityClass()` will be the current priority class of the specified process as one of the following constants: `REALTIME_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS` or `IDLE_PRIORITY_CLASS`. These constants are also used with `SetPriorityClass()` to set the desired priority class of a process.

Use caution when selecting the priority class for a process. Although it's easy to assign a process a `REALTIME` or `HIGH` priority class, this can have serious consequences for the performance of the overall system. Reread the section at the beginning of this chapter on priority classes for more information.

```

BOOL SetPriorityClass(
    HANDLE hProcess,           // process handle
    DWORD fdwPriority           // priority class flag
);

```

Accessing the command line

Besides the environment variables, the command line also provides the ability to pass information from the parent process to the child process. If the child process is expecting certain information in the command line, it can use the API function `GetCommandLine()` to obtain a pointer to the command line. The command line is available as a C string. It's terminated, as usual, by a NUL byte.

```
LPTSTR GetCommandLine(VOID);
```

First is the program name in the command line. It's followed by the parameters that were specified with the process call. Evaluating these parameters is up to the application because the Win32 API provides no additional functions for this purpose.

Getting status information on a process

Processes frequently need to determine their identity. This is required for API function calls, as a process handle or process ID. The two functions `GetCurrentProcess()` and `GetCurrentProcessId()` make this possible. `GetCurrentProcess()` returns the process handle of the current process, `GetCurrentProcessId()` returns the process ID.

```

HANDLE GetCurrentProcess(VOID);
DWORD  GetCurrentProcessId(VOID);

```

The appearance of a process, or rather that of its main window, is controlled by the parent process when it makes the `CreateProcess()` call, using a `STARTUPINFO` data structure. If a process wishes to read the contents of this structure, it can do so using `GetStartupInfo()`.

```

VOID GetStartupInfo(
    LPSTARTUPINFO lpStartupInfo // pointer to the memory for STARTUPINFO
);

```

`GetStartupInfo()` requires a pointer when it's called to a variable of the `STARTUPINFO` type. This variable receives the data from the `STARTUPINFO` structure that was specified when the process was created. When evaluating the contents of the structure, please note that not all elements of the structure represent meaningful values. The `STARTUPINFO.dwFlags` field indicates which elements of the structure have been initialized by the parent process.

Accessing foreign processes

For a process to influence another process, for example to terminate it or change its priority class, it needs a handle to that process. Most process management functions require process handles. If the process ID of the desired process is known, it's easy to obtain the handle using `OpenProcess()`.

```

HANDLE OpenProcess(
    DWORD fdwAccess,           // defines the means of access to the process
    BOOL fInherit,            // determines whether the new process handle can be
                                // inherited
    DWORD IDProcess            // process ID of the process that is to be opened
);

```

The function requires two more parameters besides the process ID of the process that you wish to access (not the current one). `fdwAccess` determines the validity of the returned process handle regarding process-function calls from the Win32 API. For this purpose, `WINBASE.H` contains several `PROCESS_...` constants. Each represents one or two process functions. For

example, if you combine `PROCESS_TERMINATE` with `PROCESS_SET_INFORMATION`, the returned process handle may be used to call the API functions `TerminateProcess()` and `SetPriorityClass()`. Other functions, such as `DuplicateHandle()` or `ReadProcessMemory()` won't work with this handle, however. If you don't want to worry about the different `PROCESS_...` constants, specify `PROCESS_ALL_ACCESS` for the `fdwAccess` parameter. This allows the returned handle to be used with all process functions.

The parameter `flInherit` determines whether the returned handle can be inherited by the child process. Specify `TRUE` if you want the handle to be inherited, otherwise `FALSE`. As usual, any handle that has been acquired in this manner should be deleted and returned to the system with `CloseHandle()`, before the process is ended.

Inter-process communication

If a parent process starts a child process to work together with it, using multitasking on the process level, it will most likely want to communicate with its child process. The system has many versatile tools for this purpose. These tools include such as memory mapped files, pipes and mailslots. The simplest form of communication, however, is exchanging messages. A process can send messages to another's main window using `SendMessage()`. Messages from the other process will arrive in the same way in which the process receives commands (messages) from the system. This form of communication is, therefore, easy to implement for the programming required.

The parent process requires a window handle to the main window in the child process to send the desired messages. It can obtain this handle using `FindWindow()`. However, the child process must have already opened its main window if the `FindWindow()` function call is to succeed. This is where we run into a problem. `CreateProcess()` immediately returns to the caller while the child process is still being constructed. If the `CreateProcess()` call is immediately followed by a `FindWindow()` call, you're likely to get an error. The child process has yet to open its window. Although it would be possible to simply keep making `FindWindow()` calls until the child process' main window has finally been opened and `FindWindow()` is finally successful, there is a more simple way to do it that also wastes fewer resources.

Instead of hassling the processor with endless `FindWindow()` calls, the parent process uses the `WaitForInputIdle()` API function. This function pauses the execution of the process that makes the function call until the specified process has completed all the messages in its message queue. This is the case only when the latter has processed all the initial messages in its message queue (including `WM_PAINT`). We know then that the main window has been opened and the parent process can make the `FindWindow()` call.

```
DWORD WaitForInputIdle(
    HANDLE hProcess, // process handle of the process that is being waited
    DWORD dwTimeout  // for maximum wait time, in milliseconds
);
```

The arguments that `WaitForInputIdle()` requires are the handle to the process and a timeout value, specified in milliseconds. If the specified process fails to process all its messages in the specified amount of time, `WaitForInputIdle()` will return a corresponding error value. If you don't wish to set a timeout value, specify the constant `INFINITE` for `dwTimeout`. `WaitForInputIdle()` returns the following function results:

- 0 if the specified process has finished all of its messages.
- The constant **WAIT_TIMEOUT** if the specified process fails to process all of its messages within the specified time.
- -1 if an error occurs during the function call.

The following is an example for using `WaitForInputIdle()`. A process produces a child process and tries to communicate with it through its message queue. It uses `WaitForInputIdle()` to wait until the child process has created its main window and processed all initial messages. Then it uses `FindWindow()` to obtain a window handle to the child process' main window and sends it a message using `SendMessage()`.

```
#define WM_DO_SOMETHING_USEFUL WM_USER + 1 // message for child process
```

```

HANDLE hChildProcess;           // process handle for the child process
HWND hChildProcessWindow;       // window handle for the child process
// create child process and wait until it has constructed
// its main window and processes all initial messages

hChildProcess = CreateProcess(...);
WaitForInputIdle( hChildProcess, INFINITE );

// child process' message queue is now empty,
// so handle to its main window can now be obtained

hChildProcessWindow = FindWindow( "ChildProcessWindowClass",
                                   "ChildProcessCaption" );

// send message to child process, if its main window was located
if( hChildProcessWindow )
    SendMessage( hChildProcessWindow, WM_DO_SOMETHING_USEFUL, 0, 0 );

```

Exchanging data

The simple exchange of messages, and thus simple commands, is often insufficient for two communicating processes. When it's necessary to transfer larger amounts of data from one process to another, use the `lParam` parameter of `SendMessage()` or `PostMessage()`. Then simply provide the other process with a pointer to the desired data. However, any such attempt, even under the best intentions, is likely to fail under Windows 95. The system keeps the address ranges for the two processes securely separate. Although the process that makes the `GetMessage()` call receives the pointer together with the message, it's only able to access its own address range and not that of the sender. This is, therefore, not a workable method for accessing the data we're trying to reach.

Therefore, Windows 95 provides a new message, `WM_COPYDATA`. It makes it possible to exchange data between processes through the message queue. `WM_COPYDATA` is not a normal message that the system simply passes to the recipient without any sort of processing. Instead, the system copies data from the address range of the sender into the address range of the recipient when the message is sent. The `lParam` parameter tells the system how many bytes are to be copied and where these are to be found. This parameter must point to a `COPYDATASTRUCT` structure when the `WM_COPYDATA` message is sent.

```

typedef struct tagCOPYDATASTRUCT {
    DWORD   dwData;           // room for message
    DWORD   cbData;           // number of bytes to be copied
    PVOID   lpData;           // pointer to the bytes to be copied
} COPYDATASTRUCT;

```

In the case that two processes are using the `WM_COPYDATA` message to transfer different data packets, the first field, `dwData`, can be used for a code or message that describes the contents of the data packet. The number of bytes in the data packet, and the number of bytes that are thus to be copied from the sender to the recipient, are specified in `cbData`. The actual data are referenced by the pointer in `lpData`. Note the specified data block must contain no further pointers to data in the sender's address range. These will not be assessable from the recipient's address range.

Once the message is sent, the system copies the specified data to the recipient's address range and replaces the content of `lpData` with a pointer to the copied data in the recipient's address range. Once the recipient has called `GetMessage()` and `DispatchMessage()`, the data reach the window function of the window whose handle was specified by the sender with `SendMessage()`. The window function will find a pointer to the received `COPYDATASTRUCT` structure in the `lParam` parameter. It can access the data through this structure. The following code sequence demonstrates the transfer of data using `WM_COPYDATA` on the part of the sender and the recipient.

```

#include <windows.h>

```

```

//— Constants —————
#define ID_FUNDATA 1                                // data packet types
//— Types —————
typedef struct tagFUNDATA                          // fun data to transfer
{                                                    // to other process
    char szFunText[ 256 ];
    int iFunNum;                                    // must not contain
    long lFunWord;                                  // any pointers!
} FUNDATA;
typedef FUNDATA *PFUNDATA;
//*****
//— the sender process
//*****
void Sender( HWND hWndSrc,                          // handle to my window
             HWND hWndDst )                        // handle to the recipient window
                                                    // in the other process
{
    FUNDATA FunData;                                // data to be sent
    COPYDATASTRUCT cds;                            // for WM_COPYDATA
    // .
    // first starts the child process in order to then use WM_COPYDATA to send
    // data
    // .
    // initialize silly data —————
    lstrcpy( FUNDATA.szFunText, "Fun! Fun! Fun!" );
    FUNDATA.iFunNum = 0;
    FUNDATA.lFunWord = 9999;
    // send silly data to window of other process using WM_COPYDATA ———
    cds.dwData = ID_FUNDATA;                        // packet code
    cds.cbData = sizeof( FUNDATA );                // length of data
    cds.lpData = &FUNDATA;                         // pointer to data
    SendMessage(hWndDst,                          // recipient window
                WM_COPYDATA,                       // message code
                (WPARAM)hWndSrc,                   // my window
                (LPARAM)&cds);                     // pointer to COPYDATASTRUCT

    // the recipient has received the message, COPYDATASTRUCT
    // can be deleted again
}
//*****
//— window function of recipient process
//*****
LRESULT WINAPI WndProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
    static FUNDATA FunData;                        // accept the received data packet
    switch( uMsg )
    {
        case WM_COPYDATA:
        {
            PCOPYDATASTRUCT pcds = ( PCOPYDATASTRUCT ) lParam;
            switch( pcds->dwData )                  // evaluate packet type
            {
                case ID_FUNDATA:

```

```

        // execute plausibility test on length of structure
        if( pcds->cbData == sizeof( FUNDATA ) )
        { // length OK, copy received data to local variable
            FUNDATA = *( PFUNDATA )pcds->lpData;
            return TRUE;
        }
        break;
        case ID_IRGENDWAS:           // if more than one type of data packet
            break;                   // are being sent this way }}
    }
    return FALSE;                   // unknown packet type
}

// message other than WM_COPYDATA -> process normally
return DefWindowProc( hWnd, uMsg, wP, lP );
}

```

The WM_COPYDATA message can be sent either with SendMessage() or PostMessage(). However, it's preferable to use SendMessage() since this message only returns once the recipient has received the message and the system has copied the specified data. PostMessage(), on the other hand, returns immediately to the caller. The caller cannot then be certain the data has already been copied. Problems can develop if the caller immediately releases the data and the recipient hasn't accepted the data yet and the system must copy the data. It's also important the recipient doesn't change or release the data. If it needs to further process the data after SendMessage() has returned to the caller (after the window function on the part of the recipient has executed return), it will first need to copy the data to a local buffer.

WM_COPYDATA can also be used to exchange data between 32-bit and 16-bit Windows applications (an otherwise difficult task to accomplish). Use this method if you're combining old 16-bit applications with new 32-bit applications.

Threads

When a process is started with a CreateProcess() call, an initial thread is also created. It embodies the execution of that process. This thread is called the *primary thread*. Any desired number of further threads can be created from within this thread. The following table summarizes the most important thread functions in the Win32 API:

Thread functions	
Function	Description
CreateThread	Creates a thread.
SetThreadPriority	Sets the priority of a specific thread.
SuspendThread	Increments the suspend-counter of the current thread and pauses its execution.
ResumeThread	Decreases the wait counter of a specific thread and, if appropriate, continues its execution.
Sleep	Pauses the execution of the current thread for a specified amount of time.
SleepEx	Special version of Sleep() used with the simultaneous use of asynchronous I/O operations.
ExitThread	Ends the current thread.
TerminateThread	Ends a specified thread in the current process from the outside.
GetCurrentThread	Returns a pseudo-handle for the current thread.
GetCurrentThreadId	Returns the thread ID of the current thread.
GetExitCodeThread	Returns the exit code for a specific thread.
GetThreadPriority	Returns the priority of a specific thread.
AttachThreadInput	Redirects the messages of a thread to another thread.

Creating threads

Besides the initial thread that is created automatically when a process is started, we can create additional threads any time within the process. The system doesn't set a fixed limit for the number of threads in a process. (You should, however, create only as many threads as you need). New threads are created using the `CreateThread()` API function, which returns a handle to the new thread upon its successful completion. This handle is used to access the thread with further calls of thread functions from the Win32 API. If the function call fails, the value `NULL` is returned. Should that happen, use `GetLastError()` to obtain the expanded error code.

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES  lpThreadAttributes,          // Win95: NULL
    DWORD                  dwStackSize,                 // size of the thread stack in
                                                         // bytes
    LPTHREAD_START_ROUTINE lpStartAddress,              // addresss of start function
    LPVOID                  lpParameter,               // argument for thread function
    DWORD                   dwCreationFlags,            // type and status of the new
                                                         // thread
    LPDWORD                 lpThreadId                  // pointer to variable that
                                                         // will accept the thread ID
);
```

lpThreadAttributes

This parameter makes it possible in Windows NT to define security attributes for the thread that is being created. Since Windows 95 doesn't incorporate the security architecture inherent in NT, you should specify `NULL` for this parameter.

dwStackSize

Each thread within a process has its own stack. Besides the return addresses, this stack is used to store the local and automatic variables of the functions that are used within the scope of this thread. The initial size of this stack in bytes can be specified with the `dwStackSize` parameter. However, you'll usually want to simply specify the value 0 for this parameter. This means the stack will be the same size as the stack of the primary thread of the process. If the stack grows beyond this size, it's automatically increased to a maximum of 1 Meg. Therefore, you won't have to worry about the stack overflowing.

lpStartAddress

This is where the start address is specified at which the execution of the new thread is to begin. This has to be a function with the `WINAPI` attribute, which receives only one argument as a 32-bit value and also returns a 32-bit value. It's up to you what type of code to use for this prototype. It doesn't matter whether you define the argument as a `LONG`, `DWORD`, `LPVOID` or another type. However, the type must be resolved into a 32 bit value when it is compiled. The following examples are valid:

```
DWORD WINAPI ThreadStartFunction( LONG lThreadParameter );
LONG WINAPI HereWeGo( LPTSTR lptstrFilename );
```

lpParameter

Represents the 32-bit value that is passed to the thread's start function when it is called.

dwCreationFlags

If the newly created thread is not be executed immediately, you must specify the constant `CREATE_SUSPENDED` for this parameter. The thread will then only begin to run once `ResumeThread()` is executed successfully. If, on the other hand, the thread is supposed to run immediately, specify 0 for this parameter.

lpThreadId

This parameter requires the address of a `LONG` variable that will contain the thread ID of the new thread. The following code sequence shows a function called `CreateAllThreads()` in which a total of 20 threads are created. They're all started using `threadPainterFunction()`. The argument for the function is a window handle, which represents the window that was previously created in the main program. The thread performs its task in this window.

```
//— Types and constants —————
#define NUM_THREADS 20

//— Prototypes —————
DWORD WINAPI threadPainterFunction( LPVOID lpStart );
void DoPaint( HWND hWnd );

//— Global variables —————
HWND hwndArray[ NUM_THREADS ];           // array containing window handles
HANDLE hPainterThreads[ NUM_THREADS ];    // array containing thread handles
DWORD dwThreadIDs[ NUM_THREADS ];         // array containing thread IDs

//— This is where the threads are created —————
void CreateAllThreads( void )
{
    int i;
    for( i = 0; i < NUM_THREADS; i++ )
        hPainterThreads[ i ] =
            CreateThread( NULL,                // lpssa
                        NULL,                // cbStack
                        threadPainterFunction, // lpStartAddr
                        hwndArray[ i ],       // lpvThreadParm
                        0,                    // fdwCreate
                        &dwThreadIDs[ i ] );  // lpIDThread
}

//— The thread function —————
DWORD WINAPI threadPainterFunction( LPVOID lpStart )
{
    DoPaint( ( HWND )lpStart );              // output in window
    return 0;                               // exit code
}
```

Exiting threads

The two different API functions available for ending threads are `ExitThread()` and `TerminateThread()`. `ExitThread()` allows a thread to end itself and `TerminateThread()` lets one thread end another thread within the same process. In either case, it's not possible to end a thread in another process.

```
VOID ExitThread(
    DWORD dwExitCode                // exit code of the thread
);
```

The stack is released with its thread. However, the handles that the thread has acquired in the course of its actions and not specifically freed again are not automatically released. Since handles are assigned to processes rather than threads, they are released only once their process is ended. Even the thread handle remains open until it is explicitly closed with `CloseHandle()` by another thread within the process or until the process is ended.

A thread doesn't necessary have to use `ExitThread()` to be terminated. Simply ending the thread function will end the thread and return value specified with `return` will be used as the thread's exit code. If `ExitThread()` is used, the exit code has to specified as a parameter as a `DWORD`. Regardless of how a thread is ended, use `GetExitCodeThread()` to determine its exit code. However, you must have a handle to the thread in question.

```

BOOL GetExitCodeThread(
    HANDLE   hThread,                // handle to the thread
    LPDWORD  lpExitCode // pointer to DWORD that will receive the exit code
);

```

In this case, the desired exit code is stored in the DWORD whose address is specified by the function's `lpExitCode` parameter. If the thread is still running, it will contain the constant `STILL_ACTIVE` following the function call. The function result tells you whether the exit code was determined. `TRUE` is returned if the exit code was determined, otherwise `FALSE` is returned.

Use `TerminateThread()` to end a thread from within another thread. As with `GetThreadExitCode()`, this requires that you have a handle to the thread you wish to end. The other argument that is required is desired exit code for the thread that you want to end. The function result shows whether the thread could be ended (`TRUE`) or not (`FALSE`).

```

BOOL TerminateThread(
    HANDLE   hThread,                // thread handle
    DWORD    dwExitCode              // exit code for the thread
);

```

Whenever possible, you should use `ExitThread()` rather than `TerminateThread()`. It represents not only the implicit termination of the thread, it represents the method the system prefers. While `ExitThread()` informs all the DLLs that are docked to the thread of its termination, `TerminateThread()` fails to do so. Depending on a DLL's implementation, this may have bitter consequences, since it's possible that the DLL is relying on this type of a message to fully complete its work. Remember, regardless of whether a thread is explicitly ended with `ExitThread()` or `TerminateThread()` or implicitly by leaving the thread function: if it's the last thread in the process, the process will also be ended.

Thread priorities

Whenever a new thread is created with `CreateThread()`, it receives the `THREAD_PRIORITY_NORMAL`. Its priority is thus equal to the base priority level of its process. However, if an application considers it necessary, it can increase or decrease a thread's priority by up to two points from the process' base priority. This is done using the `SetThreadPriority()` API function. It requires the handle of the desired thread as its first argument. This allows a thread to not only change its own priority level but also of other threads within its process.

```

BOOL SetThreadPriority(
    HANDLE   hThread,                // thread handle
    int      nPriority                // priority level of the thread
);

```

The desired priority is specified in the `nPriority` parameter. It must include one of the constants in the table on the right. Although most of these constants modify the thread priority based on the base priority of the process, `THREAD_PRIORITY_IDLE` and `THREAD_PRIORITY_TIME_CRITICAL` represent absolute priorities.

`THREAD_PRIORITY_IDLE` has a priority level of 1 and the priority level of `THREAD_PRIORITY_TIME_CRITICAL` is

15. This allows a process to create threads that, regardless of the process' priority, run at a very low or high priority level. Such threads would thus either run only when the system isn't busy doing anything else (`THREAD_PRIORITY_IDLE`) or would be ranked below realtime threads (`THREAD_PRIORITY_TIME_CRITICAL`) and be executed before all normal applications. If the function was able to set the desired priority level, it returns the value `TRUE`, otherwise `FALSE` is returned.

Thread priority	Priority increment
<code>THREAD_PRIORITY_LOWEST</code>	-2
<code>THREAD_PRIORITY_BELOW_NORMAL</code>	-1
<code>THREAD_PRIORITY_NORMAL</code>	+0
<code>THREAD_PRIORITY_ABOVE_NORMAL</code>	+1
<code>THREAD_PRIORITY_BELOW_NORMAL</code>	+ 2
<code>THREAD_PRIORITY_IDLE</code>	= 1 (see below)
<code>THREAD_PRIORITY_TIME_CRITICAL</code>	= 15 (see below)

GetThreadPriority() is used to ascertain the current priority level of a thread. The only argument it requires is the handle of the desired thread. Its return value is the priority level of the specified thread as one of the THREAD_PRIORITY_... constants from above. If an incorrect thread handle was specified, the function will return the value THREAD_PRIORITY_ERROR_RETURN. Unfortunately this function does not inform you of a thread's absolute priority between 0 and 31.

```
int GetThreadPriority(
    HANDLE hThread           // handle of the thread
);
```

Priority distribution

If we consider the priority classes of the processes, the priorities of their threads, and the automatic boosting of the foreground application by the system, we get the following priority distribution:

Process priority	Thread priority						Time critical
	Idle	Lowest	Below normal	Normal	Above normal	Highest	
Idle	1	2	3	4	5	6	15
Normal application - background	1	5	6	7	8	9	15
Normal application - foreground	1	7	8	9	10	11	15
High	1	11	12	13	14	15	15
Time-critical	16	22	23	24	25	26	31

When assigning priorities to your threads, remember the system threads for the management of the mouse, keyboard, etc., begin at priority level 12. Assigning priorities above 11 to the threads steals the system threads of processor time. However, as the table shows, you'll stay out of trouble if you use NORMAL priority processes and keep their threads below TIME CRITICAL. Remember, problems can occur with HIGH priority processes.

Thread synchronization

A thread can use SuspendThread() to pause itself or another thread if it has that thread's handle. The paused thread is stopped immediately and is ignored in scheduling until ResumeThread() is called with the handle of that thread.

```
DWORD SuspendThread(
    HANDLE hThread           // handle of the thread that is to be paused
);
```

A special suspend counter is managed for each thread. This counter is initialized at 0 when the thread is created. Each time SuspendThread() is called, this counter is incremented and the thread stopped when the counter is greater than 0. The DWORD that SuspendThread() returns as its function result represents the value before the function was called.

When the thread is to be continued with ResumeThread(), the suspend counter is lowered again. However, its execution continues only when ResumeThread() is called often enough for the counter to reach 0 again. In other words, for the function call to actually continue the thread, ResumeThread() must be called the same number of times as SuspendThread() was called. Although the very first SuspendThread() call will pause the thread, a single ResumeThread() call won't do if another SuspendThread() call was made right after the first one. In that case, the second ResumeThread() call will resume the thread's execution.

```
DWORD ResumeThread(
    HANDLE hThread           // handle of the thread that is to be resumed
);
```

If you want to resume a thread at a certain point within the program without knowing how many `SuspendThread()` calls have been made for that thread, use the following hint. Look at the function result returned by `ResumeThread()`, which represents the preceding suspend counter value. If the returned value isn't 1, keep repeating the `ResumeThread()` call. When the function finally does return the value 1, you'll know the suspend counter has reached 0 and the thread is once again running. Proceed with caution; if the first `ResumeThread()` call returns the value 0, you know that the thread wasn't even suspended in the first place. So, any subsequent attempts to bring the suspend counter to 1 by calling `ResumeThread()` will be useless. If you don't automatically break this procedure off when 0 is returned, the result is an endless loop.

While `SuspendThread()` pauses a thread for an unspecified amount of time, use `Sleep()` to set a specific duration for which the thread will remain inactive. This is useful in any situation where a thread is to run only after a certain time and has completed its task in the current time slice. For example, consider an application that is supposed to search the contents of a specific directory at regular intervals —say, every two minutes— to see whether new files have been added. When new files are found, our application will send them to a previously specified email address. The thread then pauses itself again for two minutes using `Sleep()` after which it will again check the directory.

Threads in "hibernation"

`Sleep()`, unlike `SuspendThread()`, can only be used by a thread to pause itself and not another thread. So, this function doesn't require a thread handle, only the desired pause time in milliseconds. If a sleep duration of 0 is specified, the current timeslice will end immediately but the thread won't actually be paused. Therefore, when the scheduler returns to this thread, it will continue running. `Sleep(0)` is comparable to the 16-bit function `Yield()`, which isn't supported by the Win32 API.

```
VOID Sleep(
    DWORD cMilliseconds           // "sleep duration" in milliseconds
);
```

`SleepEx()` performs the same function, however, with an added capability. This function is intended for threads that need to be paused for a specific amount of time but are to be prematurely reactivated when a certain event occurs. This is not just any event but specifically the end of an asynchronous I/O operation that was started before the `SleepEx()` call with either a `ReadFileEx()` or `WriteFileEx()` call. `ReadFileEx()` and `WriteFileEx()` return to the caller immediately while the I/O operation that was started is still underway. They indicate the completion of the operation with a callback function, whose address they obtain from the caller. However, the callback function can only be executed if the thread is not currently paused with `Sleep()`. The asynchronicity of `WriteFileEx()` and `ReadFileEx()` would therefore be lost if you were to use `Sleep()` and so the thread was not to wake up before the I/O operation was completed.

```
VOID Sleep(
    DWORD cMilliseconds           // "sleep duration" in milliseconds
    BOOL fAlertable                // is previous asynchronous I/O operation completed
);
```

So, if you want to use the capabilities of `Sleep()` use a more versatile version called `SleepEx()`. If you specify the value `TRUE` for `fAlertable` when you make the function call, `SleepEx()` will return before the specified sleep duration is up when the previously started asynchronous I/O operation has been completed. If `SleepEx()` returns the function result `WAIT_IO_COMPLETION`, you know that this is what has happened. However, if it returns the value 0, the specified sleep duration elapsed before the I/O operation was completed.

Getting status information on threads

Two different functions are available that allow you to obtain information on the identity of the current process. Also, two corresponding functions are available that give you the same information on the current thread. These functions are `GetCurrentThread()` and `GetCurrentThreadId()`. The first provides the thread handle of the current thread. `GetCurrentThreadId()` provides its thread ID. Depending on the API thread function, you'll need either one of these pieces of thread identification.

```
HANDLE GetCurrentThread(VOID);    // returns the thread handle
DWORD GetCurrentThreadId(VOID);   // returns the thread ID
```

Control messages

One of the major problems with Win16, and consequently Windows 3.x, resulted from the fact there was only one message queue for all active applications. If one application crashed and failed to remove its message from the queue, all the following messages in the message queue basically got stuck and were inaccessible to their applications. The system subsequently locked up and usually had to be restarted.

To avoid these problems, Windows 95 provides each thread with its own message queue. If an application locks up, other threads can still use GetMessage() to get their messages from their own queues. Any thread that opens a window must also get that window's messages using GetMessage() or PeekMessage(). Another thread won't get the messages, since these two functions only access the message queue of the thread that calls them.

However, situations may occur in which the messages sent to a window need to be read by another thread and not by the window's creator. Use AttachThreadInput() to redirect the message queue to another thread.

```

BOOL AttachThreadInput(
    DWORD idAttach,    // thread ID whose message queue is being redirected
    DWORD idAttachTo,  // thread ID that is receiving the other's messages
    BOOL fAttach       // attach or detach the threads?
);

```

The first argument the function requires is the thread ID of the thread whose message queue is being redirected. The second parameter determines which thread will receive the redirected messages. Again, the function requires the thread ID of the desired thread. The third parameter determines whether the message queue of the first thread will be redirected to the second one or whether it will be detached from it. If you wish to redirect the queue, specify TRUE. Specify FALSE if you wish to reverse a redirection you've previously made.

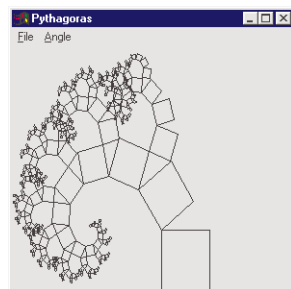
Sample Programs

Before continuing with the "synchronization" topic, we'll introduce two programs. PYTHAGO.C shows the structure of a program in which the foreground thread, which is responsible for the interaction with the user, is disengaged from the background thread that does the actual work. The second program is based on MASTER32.C. It shows how multitasking is used on the process level in conjunction with SLAVE32.C.

Multitasking with foreground and background threads

The Pythagorean theorem ($a^2 + b^2 = c^2$) describes the relationship between the sides of a right angle triangle. PYTHAGO.C displays this theorem by drawing the Pythagorean tree. This tree is formed when you take the sides of a right triangle and draw a square above each side of the triangle with sides the length of that side. Based on the hypotenuse (base side) with its given length and a given angle, you get two squares over the Catheteus. The Catheteus themselves are the hypotenuse for a new right angled triangle. So, from one right angled triangle you get two, from those two four, and so on until the Pythagorean tree appears.

Through teamwork between MASTER32 and SLAVE32, any desired number of Pythagorean trees can be drawn



The Pythagorean tree in this program serves only to show the interplay between a foreground and background thread. We could also draw any other formation or perhaps simply resort to the output of randomly placed lines. However, we wanted to show something with a little more content. So, let's look at PYTHAGO.C. This program not only draws the Pythagorean tree but also provides for user interaction.

In the application's WinMain() startup function we first create a window class with the name "Pythagoras". Then link it with a menu from the PYTHAGOR.RC resource file. Then create and display a

window of this class. Enter the message queue with `GetMessage()` and `DispatchMessage()`. This way the messages that are received reach the window function of the class with the name `WndProc()`.

Several global variables are initialized with the `WM_CREATE` call. The first are the fields of the global variable `PP` of the type `PYTHOPARMS` that will receive the corner data of the tree that will be drawn. These are, primarily, the angle of the right angled triangle in `fAlpha` and the color in `crColor`. Nothing more happens until a `WM_COMMAND` message is received to indicate the user has clicked the New menu. The program responds by displaying a dialog box in which the user specifies the angle and the color for the Pythagorean tree that the program will draw. The dialog box is controlled by the dialog function `DialogProc()`. This dialog function is not especially important here since it is a conventional dialog function.

After the user closes the dialog box, the code in `WndProc()` checks whether the dialog box was closed with the OK button. If so, then the application begins drawing the tree. This task is to be performed by a background thread whose handle is stored in the global variable `hThread`. The program code therefore first checks whether `hThread` still contains `NULL`, the value with which `WM_CREATE` initialized the variable. If it contains a different value, the user has already started the drawing process. In that case, the background thread that was created before must be terminated by setting the global variable `bTerminateThread` to `TRUE`. Together with the global variable `bStartAgain`, this variable makes communication between the background thread (which is drawing the tree) and the foreground thread (which allows for user interaction) possible.

When the background thread detects that `bTerminateThread` has been set to `TRUE`, it stops its execution. The code in `WndProc()` waits for this to happen by calling the API function `WaitForSingleObject()`. (This function will be discussed in detail in the following sections in relation to the topic of “synchronization”.) The function is given the handle of the background thread from the global variable `hThread`. This means the function must wait until the specified thread has been ended. It can only return to the caller only when the specified thread has been ended.

Once the function has returned, the program code in `WndProc()` can be certain the background thread no longer exists. This is very important because the background thread will be started again. If the background thread was still running, we’d suddenly have two background threads. This would definitely lead to some unwelcome surprises since the program is not designed to work with two background threads. First, `Thread` is set to `NULL`. Then the `hwnd` field in the global `PP` variable is assigned the handle of the application window. The background thread will later be able to tell from this variable to which window it must send its output when it is drawing the tree. The window’s contents are deleted in the next step so any tree that may have been drawn before will disappear from the screen.

Two global variables `bTerminateThread` and `bStartAgain` are also set to `FALSE` so the new background thread won’t immediately quit. Then the API function `CreateThread()` is used to create the background thread. The thread handle that is returned is stored in the global variable `hThread`. From this point, two threads exist within the application. These threads are the foreground thread, which is still within the `WndProc()`, and the background thread that was just created and is now beginning its execution with `threadPainterFunction()`.

`threadPainterFunction()` first decreases its own priority level by one point. This emphasizes its role as background thread (priority: `THREAD_PRIORITY_BELOW_NORMAL`). Then it calls various functions from the `TURTLE32.C` module that are used to draw the Pythagorean tree. These functions are linked to the Pythagoras application with `Make-File`. Turtle emulates a simple graphics system, like the one used in the `LOGO` language. Through a series of operations (here implemented as functions in `TURTLE32.C`), this “turtle” can crawl across the screen, leaving a visible trail. If you want to draw a line in a certain direction, all you need to do is rotate the turtle so it is facing the right way and tell it how far to crawl.

Although in this implementation in `TURTLE32.C` you won’t actually see a turtle on your screen, you will see the lines marking its path. Without discussing the functions in `TURTLE32.C` in detail, we’d like to point to an interesting characteristic of the module that is also used in `threadPainterFunction()`. This is the ability to scale the drawing output of the `TURTLE` module.

In `threadPainterFunction()`, the tree is first drawn without actually displaying it on the screen. We need to first determine the tree’s dimensions so the `TURTLE` module can scale its output in the window correctly. When `Pythagoras()` is called the second time, a parameter lets the tree be drawn visibly, and the `TURTLE` module scales it to fit perfectly into the window.

This happens in a loop in which the tree is redrawn, the window background is deleted and the loop is executed again. The loop ends only when the function discovers the foreground thread has set the global variable `bTerminateThread` to `TRUE`.

This indicates the background thread is to be terminated. Since threads share the global variables of their processes, you have a very simple and efficient method of letting threads communicate with one another.

The actual drawing function, `Pythagoras()`, also checks the two global variables `bTerminateThread` and `bStartAgain` and returns to the caller when it discovers that one of these has been set to `TRUE`. Keep in mind that `Pythagoras()` is a recursive function. It first draws the specified hypotenuse square and then executes a recursion on the left and then a recursion on the right. This is why the left branches always appear on the screen before the right branches of the tree. The recursion ends at a predetermined side length (specified in the constant `MIN_EDGELENGTH`) so it doesn't continue to infinity. However, the recursion ends immediately if one of the two global variables `bTerminateThread` or `bStartAgain` is set to `TRUE` by the foreground thread.

`bStartAgain`, unlike `bTerminateThread`, only ends the `Pythagoras()` function. Then, the loop is continued rather than ending `threadPainterFunction()` so the window background is deleted and a new tree is drawn with the `Pythagoras()` function. After all, the dialog function doesn't set `bStartAgain` to end the application but rather because it is responding to a `WM_SIZE` message. This means that the user has resized the window so the background thread needs to redraw the tree to fit the window's new dimensions.

This application can also be used to demonstrate the system's dynamic priority boosting quite nicely. Simply start several instances of the application and compare how fast the trees are drawn in the different windows. You'll find that the tree in the active foreground window will always be drawn somewhat faster than those in the background windows. Also observe what happens when you move a window or start another application. This will give you a better feel for priority management than we can convey on paper here.

Multitasking with multiple processes

This section talks about multitasking with processes. The programs in this section also draw Pythagorean trees. The difference this time, however, is that the programs draw any desired number of Pythagorean trees and not just one. The program is based on `PYTHAGO.C` and is called `SLAVE32.C`. It's only different in a few, but important, areas from `PYTHAGO`. Unlike `PYTHAGO`, `SLAVE32` doesn't contain a menu through which the user can control the application. Instead, `SLAVE32` uses `MASTER32` to allow for user interaction. Whenever the user wishes to start another Pythagorean tree, `MASTER32` uses `CreateProcess()` to start another instance of `SLAVE32` to render the desired tree on the screen.

The interesting point about this example is the manner in which `MASTER32` and `SLAVE32` communicate with each other. Information that determines how the tree appears, such as the angle of the Pythagorean tree and the color, are passed from the master to the slave. The message that the program will be ended, when the user clicks the Close button in the slave's title bar, is passed from the slave to the master.

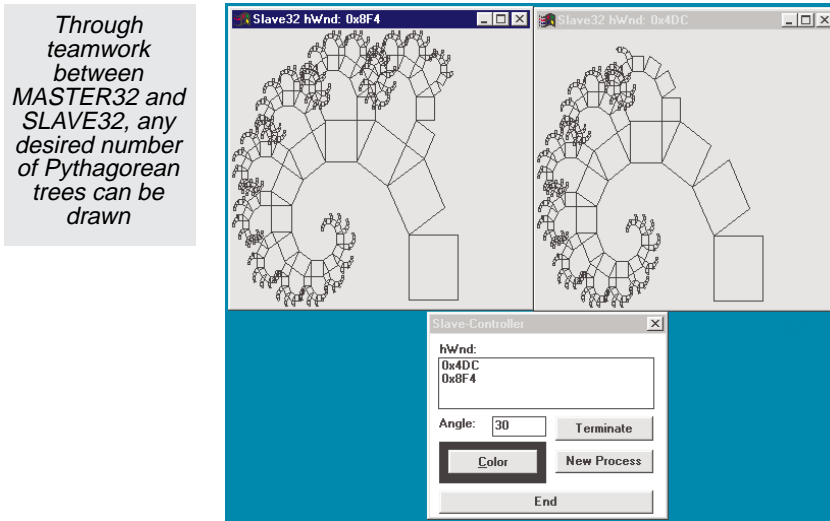
The information is exchanged between the two programs as messages that are sent to the main window of the other program. This means they must each know the handle of the other's main window. The master process must first start the slave process using the API function `CreateProcess()`. Before it calls this function, it creates an environment variable called `Window` in the block containing the environment variables for the slave process. This variable is assigned the value of the master process' window handle as an ANSI string. After starting, the slave process searches its environment variables for an entry called "Window" and so obtains the handle to the master process. Then it uses `SendMessage()` to send a message with the name `PM_PROCESSCREATED` to this window. It assigns its window handle to the message in the `LPARAM`. This lets the master know where to address its messages to the slave. This makes bidirectional communication possible.

`PM_PROCESSCREATED` is only one of six private messages that are passed between the master and slave. They're defined in the `PRIVAT.H` include file. This file is linked into both programs so they "speak the same language."

You'll find the following program(s) on the companion CD-ROM



`PYTHAGO.C` (C listing)
`RESOURCE.H` (C listing)
`PYTHAGO.RC` (C listing)
`TURTLE32.H` (C listing)
`TURTLE32.C` (C listing)



After it receives `PM_PROCESSCREATED`, the master sends the slave `PM_ANGLE` first and then `PM_COLOR`. It writes the color (or angle) specified by the user into the `LPARAM` of these messages. However, the slave only really starts its task after the master has also sends it a `PM_CREATETHREAD` message. The slave to this point consists only of one thread (its primary thread). It was created when the slave was started. Once the slave receives the `PM_CREATETHREAD` message, it creates the actual paint thread, which draws the Pythagorean tree. The code used to render the tree in `SLAVE32` may be familiar from the program code of `PYTHAGO.C`. It's basically the same as the functions `threadPainterFunction()` and `Pythagoras()`. So, two threads will be running for each slave process. One thread receives the messages and a second thread that is responsible for drawing the Pythagorean tree.

A slave process will continue to render Pythagorean trees until the user terminates it by clicking its Close button. If this occurs, the slave process first sends a `PM_PROCESSHASEXIT` message to its master, specifying its window handle in the message's `LPARAM`. This way the master knows which slave has been terminated. It can then remove the appropriate entry from the list box containing the window handles of the active slaves.

The master can also terminate slaves. It does so when it's terminated by the user or when the user selects a slave from the list box and then clicks the Terminate button. In the latter case, the master sends a `PM_EXITPROCESS` message to the selected process(es). Upon receiving this message, the slave first terminates the drawing thread and then the primary thread, right after sending a `PM_PROCESSHASEXIT` message to the master.

As you can see, it's possible to establish bidirectional communication between different processes with relative ease by using messages. You may find this solution useful in many different programming situations.

You'll find the following program(s) on the companion CD-ROM



PRIVAT.H (C listing)
MASTER32.C (C listing)
SLAVE32.C (C listing)
RESOURCE.H (C listing)

Synchronization

Whenever multiple processes or threads are working together on a task, it becomes necessary to coordinate, or *synchronize*, their work. Consider the classic example of the foreground thread that interacts with the user and the background thread that is to print the data entered by the user. What happens if no data is available, what if the user suddenly tells the foreground thread to stop the printout? What is needed is a reliable means of communication between the threads (see the previous Section for more information on communication between the threads). The previous program examples used global variables to facilitate the communication between a foreground and its background thread, however, this solution won't solve all programming problems.

This solution especially won't work in a *race condition*. This occurs when multiple threads are competing for a resource that only one thread can use at a time. Here, resources don't refer to the elements of a resource file. Instead they refer to shared global and static variables, files, handles and other operating that don't work well when threads are competing for them.

Race conditions

Let's use a file that is being read by two threads at the same time for an example:

```
#include <windows.h>

typedef struct tagFILESETS                                // sets of data in the file
{
    int i;
    long l;
} FILESETS;

//— global variables —————

HANDLE g_hFile;                                          // handle of the file
HANDLE g_hThread[ 2 ];                                  // handle of the two threads

//— prototypes —————
DWORD WINAPI threadRead( LPVOID lpParms );              // prototype

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR     lpszCmdLine,
                   int       nCmdShow)
{
    g_hFile = CreateFile( "testfile.dat",                // open file
                          GENERIC_READ,
                          FILE_SHARE_READ,
                          NULL,
                          OPEN_EXISTING,
                          0L,
                          NULL );

    // create first thread and read 100th data
    g_hThread[0] = CreateThread( NULL, 0L, threadRead, ( LPVOID )100, 0L, NULL );

    // create second thread and read 200th data
    g_hThread[1] = CreateThread( NULL, 0L, threadRead, ( LPVOID )200, 0L, NULL );

    // .
    // .
    // .

    CloseHandle( g_hThread[0] );                        // delete the two thread objects
    CloseHandle( g_hThread[1] );
    CloseHandle( g_hFile );                             // close file
    return 0;
}

//— threadRead() reads a data set from the

DWORD WINAPI threadRead( LPVOID lpParms )
```

```

{
    FILESETS filesets;
    DWORD dwRead;
    LONG lRecord = ( LONG )lpParms;

    // move file pointer to the desired data set -----
    SetFilePointer( g_hFile, lRecord * sizeof( FILESETS ), NULL, FILE_BEGIN );

    // if the system switches tasks between these two threads at this point,
    // it'll go BOINK!

    // read data set -----
    ReadFile( g_hFile, &FILESETS, sizeof( FILESETS ), &dwRead, NULL );
    return 0;
}

```

This example looks quite harmless if you quickly glance through it. A file is opened in the main program body. Then two threads are created with `threadLesen` as their start function. The number of the data set that is to be read is provided as an argument. The first thread is given the 100th and the second thread is given the 200th data set. Since the application is to run on a PC with only one processor, execution will cycle between these two threads. This is where we'll run into our problem.

Let's imagine that the first thread has just moved the data pointer to the 100th data set when its execution is interrupted and the processor switches to the second thread. The latter moves the pointer to the 200th data set and reads it. The scheduler then decides that it's time to switch back to thread 1. This thread will continue by trying to read the 100th data set. However, the data pointer is now where thread 2 has set it, just after the 200th data set. Without its knowledge, thread 1 has thus read a completely different data set than was specified. It's easy to see that this won't exactly help the smooth and flawless operation of the application.

Although this scenario might seem a little far fetched, such problems do occur frequently when several threads are accessing a resource at the same time. The following is another example:

Your program manages a ring buffer like the one Windows uses internally to store messages. One or more threads send messages to the ring buffer. Other threads read from the ring buffer. These threads use `RpPutMessage()` and `RpGetMessage()`, which manage the ring buffer internally as an array and control access to the buffer with two pseudo-pointers named `iRpFirst` and `iRpLast`. `iRpFirst` will contain the index of the array element with the next message, which hasn't yet been read, while `iRpLast` points to the array element in which the next incoming message shall be stored.

```

//— types and constants -----

typedef LONG MES;                                     // ring buffer elements
#define RINGBUFFER_LEN 512                             // ring buffer size

//— global variables -----

MES RingBuffer[ RINGBUFFER_LEN ];                     // the actual ring
int iRpFirst, iRpLast;                                // index for Put and Get

//— RpPutMessage() deposits a message in the ring buffer -----
void RpPutMessage( MES m )
{
    RingBuffer[ iRpLast ] = m;                         // deposit message in ring buffer

    // if the system switches tasks between these two threads at this point,
    // it'll go BOINK!, because g_iRpLast has not yet been

```

```

if( ++iRpLast == RINGBUFFER_LEN )           // iRpLast to next slot
    g_iRpLast = 0;                          // start again at first slot
}

```

Like the programs in the first section, switching between tasks in `RpPutMessage()` or `RpGetMessage()` can cause serious trouble for the program. Let's say `RpPutMessage()` is called by a thread that wants to add a message to the end of the ring buffer. `RpPutMessage()` first quite harmlessly deposits this message in the array element to which `iRpLast` points. This, however, is where the trouble starts. Now the scheduler switches to the other thread, which also happens to call `RpPutMessage()`. Although the previous function had enough time to deposit the message in the array, it didn't get around to incrementing `iRpLast` before the scheduler switched tasks. The result is that the second function is stored in the same array element, thereby overwriting the first. This is less than ideal result.

However, it becomes worse. After the next message is deposited in the array, `iRpLast` is incremented and the function is ended. The thread continues its execution until the scheduler decides to switch back to the first thread. This occurs shortly thereafter. The latter continues where it left off and increments `iRpLast` in the `RpPutMessage()` function. This now causes an array element to be skipped entirely; otherwise, a message is written to it. Our blunder now not only resulted in a lost message. We'll encounter a message when the array is read again later with `RpGetMessage()` that was written to the array previously. This message is completely unrelated to the current data. It becomes quite clear that it is essential to synchronize these events, especially since a similar scenario can also occur with `RpGetMessage()`.

These types of race conditions are neither new nor unique to Windows. They're found in all operating systems that support multitasking. Programmers first attempted to solve these problems in the 1960s. Several solutions were found that remain applicable today. One answer found to the problem of race conditions was using *synchronization objects*. They allow threads to synchronize their access to shared resources, or to *serialize* this access. Essentially, make certain only one thread will access the resource at any time while the others are required to wait.

Synchronization objects in Win32

The Win32 API's answer to the problem of race conditions is a series of synchronization objects and accompanying API functions. Use these objects and functions to synchronize threads. Some of these objects can only be used within a process. Other objects can also be used to synchronize threads in different processes (more about that later).

The following table shows the synchronization objects provided by the Win32 API and the API functions with which they are created:

Object	Created by
Mutex	CreateMutex() or OpenMutex()
Semaphore	CreateSemaphore() or OpenSemaphore()
Critical Section	InitializeCriticalSection()
Event	CreateEvent() or OpenEvent()
Prozeß	CreateProcess() or OpenProcess()
Thread	CreateThread() or OpenThread()
Change-Notification	FindFirstChangeNotification()
Concole input	CreateFile(_CONIN\$") or GetStdHandle()

These synchronization objects have different purposes and are therefore used in very different situations.

- **Mutexes, Semaphores and Critical Sections**
are designed to avoid race conditions like those described earlier.
- **Events**
are used wherever a thread is to wait for a specific event, such as the completion of a calculation by another thread.

- Processes and Threads
are really not synchronization objects but can be used for synchronization purposes, specifically when a thread is to wait for the termination of a process or thread before it continues with its own execution.
- Change-Notifications and Console Input
represent a special type of event, with which a thread can be notified of changes within a directory or the availability of input. These objects are examined in our chapter on file operations and console applications.

Synchronization isn't created by the mere existence of these objects. Instead, the threads need to apply one of several wait functions to such an object. We talk about five wait functions in this chapter. The simplest of these is `WaitForSingleObject()`. The caller must provide the handle of a synchronization object to `WaitForSingleObject()`. It will only return to the caller once the object enters a "signalized" state. If the object remains in a nonsignalized state, the thread will be blocked from scheduling. Only once another, nonblocked thread has put the object in a signalized state can the waiting thread resume its execution.

Mutexes

A mutex is one of the most basic synchronization objects and are the objects most frequently used to avoid race conditions. The two problems illustrated above can also be easily solved using mutexes. The term "mutex" derives from "mutual exclusion." Mutex assumes one of only two states:

1. Nonsignalized (in possession of a thread)
2. Signalized ("free" of a thread)

To remember this, consider a mutex a flag (a signal). If the flag is risen, the mutex is free but if the flag is lowered, the mutex is engaged.

Like the other synchronization objects, mutexes must first be created. The Win32 API provides the special function `CreateMutex()` for this purpose. It provides its caller with a handle to the mutex that has been created. You'll normally want to store this handle in global variable so all threads within a process have access to it. The most simple wait function to acquire a mutex that a thread can use is `WaitForSingleObject()`. The thread provides this function with the mutex's handle. The function then checks whether the mutex has already been acquired by another thread. If not (the mutex is signalized), the thread that made the function call receives the mutex and the mutex is switched into the nonsignalized state. In this case, the wait function returns immediately to its caller so the latter can continue with its execution.

If the scheduler then switches to another thread that also wants to acquire this mutex, another wait function call will be made using the handle of that mutex. However, since the mutex has already been acquired by another thread (and so is not free), it is not available for this thread. The thread is therefore blocked and for the time being remains "unused." It isn't aware of this since the wait function simply doesn't return. The thread remains in this state until the current owner of the mutex frees the mutex with `ReleaseMutex()`. This function indicates that the thread no longer needs the mutex and releases it so another thread can acquire it.

That's exactly what happens...when the mutex is freed, the previously blocked thread that has been waiting for the mutex is again included by the scheduler. When it receives processor time again, it acquires the mutex. Only then does the wait function return to the thread. Now any other threads that make a wait function call will be blocked until this thread releases the mutex with `ReleaseMutex()`, even the thread that had the mutex before this one.

Win32 mutex functions	
Function	Description
<code>CreateMutex</code>	Creates a new mutex or opens an existing mutex.
<code>OpenMutex</code>	Returns a handle to an existing mutex.
<code>ReleaseMutex</code>	Releases a mutex that has been acquired.
<code>Wait...</code>	Threads can use the wait function to acquire a mutex.
<code>CloseHandle</code>	Deletes a mutex object.

However, a mutex is of little use if it is not implemented consistently. It's important to know when, where and who should acquire the mutex. Let's again refer to the example with the ring buffer and the two functions RpPutMessage() and RpGetMessage(). We'll first look at RpPutMessage(). We've seen that RpPutMessage() must only be called by one thread at a time to avoid problems with updating the array. The function call therefore must be serialized through a mutex. This is best done within the function itself, right at its start.

```
//— types and constants —————

typedef LONG MES;                                // ring buffer elements
#define RINGBUFFER_LEN 512                        // ring buffer size

//— global variables —————

MES Ringbuffer[ RINGBUFFER_LEN ];                // the actual ring buffer
int iRpFirst, iRpLast;                           // index for Put and Get

//— RpPutMessage() deposits a message in the ring buffer —————
#include <windows.h>

//— types and constants —————

typedef LONG MES;                                // type of ring buffer elements
#define RINGBUFFER_LEN 512                        // ring buffer size

//—global variables —————

HANDLE g_hMutexRpPut;                            // mutex for RpPutMessage
HANDLE g_hMutexRpGet;                            // mutex for RpGetMessage

MES g_Ringbuffer[ RINGBUFFER_LEN ];              // the actual ring buffer
int g_iRpFirst, g_iRpLast;                       // Put and Get Index

//— start function —————

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR     lpCmdLine,
                   int       nCmdShow )
{
    g_iRpFirst = g_iRpLast = 0;                   // ring buffer is empty for now

    // create nonsignalized mutexes for Put and Get
    g_hMutexRpPut = CreateMutex( NULL, FALSE, NULL );
    g_hMutexRpGet = CreateMutex( NULL, FALSE, NULL );

    // .
    // threads are created that write messages to the ring buffer using
    // RpPutMessage(), while others read them with RpGetMessage()
    // .

    CloseHandle( g_hMutexRpPut );                 // delete mutex again
    CloseHandle( g_hMutexRpGet );
```

```

    return 0;
}

//— RpPutMessage() deposits a message in the ring buffer ———
BOOL RpPutMessage( MES m )
{
    BOOL bRetVal;
    CRITICAL_SECTION csRpLast;

    // wait for the mutex, in case another thread has taken control
    // of the mutex

    WaitForSingleObject( g_hMutexRpPut, INFINITE );

    // no other thread gets to this point if the mutex isn't freed again
    if( g_iRpFirst != ( g_iRpLast + 1 ) % RINGBUFFER_LEN )
    {
        // the ring buffer still has space for another message
        g_Ringbuffer[ g_iRpLast ] = m;

        EnterCriticalSection( &csRpLast );           // so it doesn't go BOINK!

        if( ++g_iRpLast == RINGBUFFER_LEN )
            // switching tasks at this point could be fatal
            g_iRpLast = 0;

        LeaveCriticalSection( &csRpLast );

        bRetVal = TRUE;
    }
    else bRetVal = FALSE;

    // release mutex again, so other threads can call the function

    ReleaseMutex( g_hMutexRpPut );

    return bRetVal;
}

```

When a thread calls this function, it first tries to acquire the mutex `g_hMutexRpPut` for the thread. If another thread is already within the function the mutex is not free. The thread is therefore blocked. However, it is only blocked until the thread that is currently within the function has reached the end of the function and releases the mutex with `ReleaseMutex()`. Only then can the waiting thread enter the function. This ensures that only one thread will be within the function at any one time. This avoids problems with updating the internal structures. The prerequisite for the success of this method is that the mutex is created at the start of the program with `CreateMutex()` and that its handle is then stored in the global variable `g_hMutexRpPut`.

`RpGetMessage()` also needs to be protected in the same way since the function faces the same potential dilemma of being open to any thread at any given time. However, we now need to decide an important question: Should `RpGetMessage()` be protected with the same mutex that is used for `RpPutMessage()` or with another one? If the same mutex is used, then only one of the two functions can be executed at any one time. If one thread calls `RpGetMessage()`, then no other thread will be able to run `RpPutMessage()` during this time, and vice versa.

If, on the other hand, a separate mutex is used for `RpGetMessage()`, a thread can call `RpPutMessage()` while another runs `RpGetMessage()` at the same time. This is essentially the preferred approach. Remember, the more you serialize the program, the more you're lose the advantages of multitasking. This approach could be taken to such an extreme the application is

completely serialized and only one thread is executed at any one time. Considering the cost of processor time dedicated to switching between tasks, this type of application may run slower than one consisting of only one single thread.

Although this may not be an issue with our little program, we still need to decide whether `RpGetMessage()` and `RpPutMessage()` should be able to be executed at the same time. Is there any instant at which the two functions could conflict with each other during a switch between tasks? At a first glance, the answer seems to be no, since each is equipped with its index, and therefore does not modify the other's. This means the functions should be able to function with a separate mutex.

Although `RpGetMessage()` accesses the index `iRpLast` to check whether there is still a message in the buffer that hasn't been read, it does not modify the index. This is the requirement that keeps the two functions from conflicting with each other. If it were to change the index, we'd have an entirely different situation. The rule of thumb is that multiple threads may access a shared resource simultaneously without conflict if they only *read* the resource. Access conflicts arise only when they write to the resource at the same time. This is not the case here. You might assume the two functions would never interfere with one another. This is indeed true for most of the commands within the functions. However, a second, closer look reveals a potential problem. Both contain a command that could imbalance the delicate cooperation between Put and Get in their shared use of the ring buffer.

Although we're only speaking of an If command and its then counterpart, looks can be deceiving. The problem is encountered in the two functions after they've inserted or rather read a "-" character. After this the appropriate index, `g_iRpLast` or `g_iRpFirst`, must also be modified accordingly. In `RpPutMessage()`, for example, the following line:

```
if( ++g_iRpLast == RINGBUFFER_LEN )
    // switching tasks at this point could be deadly
    g_iRpLast = 0;
```

Here `g_iRpLast` is first incremented so it points to the next free slot for a character. Then a check is made to see whether the end of the buffer has been passed. If so, the index must be set to 0. Here, between the incrementation and the decision to make a correction, that the seemingly unthinkable can happen: a switch to another task in which `RpPutMessage()` is called. This function then reads `g_iRpLast`, a variable that is currently not up to date, since `RpGetMessage()` was just going to set it to 0, but was prevented from doing so by the switch to another task.

You must avoid this scenario. Whenever a task reads a resource, it must be able to rely on the fact that no other task has left it in an undefined state (i.e., that it is up to date). We could reconsider the situation and decide that this is reason enough to make certain the functions don't run at the same time by using the same mutex for both. However, there's another solution to the problem. Notice the If statement in question in the listing of `RpPutMessage()` above is placed in a critical section. There it only applies to the If command and its then statement. This ensures that the scheduler doesn't switch to another task during this command, thereby avoiding this unlikely, but lethal scenario. The same solution is used in `RpGetMessage()`.

Although the above considerations may seem a little nit-picky, it's important to mentally run through such scenarios when you're developing multitasking code. If you miss a spot like this, your application is likely to crash in totally unexpected times. Also, it will be nearly impossible to trace the problem, which is probably one of the worst bugs that can plague a program.

```
//— RpGetMessage() reads the next message that hasn't been read

MES RpGetMessage( void )
{
    MES m;
    CRITICAL_SECTION csRpFirst

    // wait for Mutex, in case another thread has already acquired it

    WaitForSingleObject( g_hMutexRpGet, INFINITE );

    // no other thread will reach this point if the mutex is in use
```



```

if( g_iRpFirst != g_iRpLast )           // is there an entry in the buffer?
{
    m = g_Ringbuffer[ g_iRpFirst ];      // read message

    EnterCriticalSection( &csRpFirst ); // so it doesn't go BOINK!

    if( ++g_iRpFirst == RINGBUFFER_LEN )
        // switching tasks at this point could be deadly
        g_iRpFirst = 0;

    LeaveCriticalSection( &csRpFirst );
}
else m = -1L;                           // no message in the ring buffer

// release Mutex so other threads can call the function

ReleaseMutex( g_hMutexRpGet );

return m;
}

```

By looking at this function you may notice another interesting problem. The mutex is released before the return command that ends the function is executed. (This must occur since return ends the function and ReleaseMutex() could not be executed after that). So if we switch to another task just before return, and another thread runs the function, is it possible that the new thread could destroy the function result *m* of the first thread? No, since *m* is a local variable, and was therefore placed on the thread's stack. Since each thread possesses its stack, we also switch to another stack when we switch tasks. Therefore, when the return command of the first thread is finally executed after another switch between tasks, it will find the same value in the local variable *m* as before.

Mutex functions

Mutexes are created using CreateMutex(). If you only need the mutex within the current process, you can ignore the function's first and last parameters and simply specify NULL. If you want the mutex to be immediately acquired by the thread that has called the function, specify TRUE for bInitialOwner. Otherwise, specify FALSE.

```

hHANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,    // Win95: NULL
    BOOL                  bInitialOwner,        // assign mutex to caller immediately?
    LPCTSTR               lpName                // pointer to string containing the name of the mutex
);

```

If the specified mutex object was successfully created, the function will return a handle to the mutex. NULL is returned if an error has occurred. In this case you can use GetLastError() to obtain a more detailed error code. An existing mutex object can be opened with OpenMutex(). However, this is only useful if the mutex is to be used for interprocess communication (see the section on Synchronization between processes).

```

HANDLE OpenMutex(
    DWORD    fdwAccess,           // desired access to the mutex object
    BOOL     fInherit,           // TRUE, if the mutex handle can be inherited
    LPCTSTR  lpszMutexName       // pointer to the string with the name of the
                                // mutex object
);

```

You can acquire a mutex object using one of the wait functions. `ReleaseMutex()` is then used to release the mutex object. The handle of the desired mutex object is all that this function requires. However, this function will only succeed if the specified mutex is currently in the possession of the thread making the function call. No other thread may call `ReleaseMutex()` while it does not possess the mutex.

If a thread calls a wait function with the mutex handle several times consecutively, only the first call will be blocked if the mutex has already been acquired by another thread. When the thread has acquired the mutex, all the other wait calls will immediately return to the caller. However the thread must also call `ReleaseMutex()` for each successful wait call, since the mutex keeps an internal counter keeping track of the number of wait function calls and `ReleaseMutex()` calls. Only once the counter is even again, can another thread acquire the mutex.

```
BOOL ReleaseMutex(
    HANDLE hMutex    // handle of the Mutex object that is to be released
);
```

Mutexes should be deleted before the process is ended. This is true for all kernel objects created by the Win32 API during the execution of a process. Although this will be done automatically when the process is ended, it is always better to explicitly release such objects whenever they are no longer needed. However, Win32 API provides no special functions for releasing mutexes. Instead, use the general function `CloseHandle()`. It can be used to delete all types of kernel objects. Simply pass the handle of the mutex you wish to delete to the function. If the handle you specified was still valid, the function will return the value `TRUE`.

```
BOOL CloseHandle(
    HANDLE hObject    // handle of the semaphore
);
```

Caution, deadlock

Mutexes are a simple and comfortable means for avoiding race conditions. However, the use of mutexes also has its pitfalls. The situation always becomes more critical when multiple mutexes are being used within an application and several threads are trying to acquire them. Picture the following scenario: Your program consists of two threads, T1 and T2. You've created two mutexes, M1 and M2, to coordinate the access to different resources. Thread T1 first acquires M1 and thread T2 acquires the mutex M2. Before thread T1 releases M1, it tries to acquire the mutex M2. Thread T2 does the same. It only releases its mutex M2 once it has acquired M1. A seemingly harmless situation has very quickly ended in a classic deadlock — a state in which both threads are waiting for each other and neither makes any progress.

If one of the two would release its mutex, for example thread T1 releasing its mutex M1, everything would continue problem free. This would allow T2 to acquire M1. It shortly thereafter releases M2. Thread T1 would then no longer be blocked from acquiring the mutex M2. However, if neither of the threads releases its mutex, this knot won't unravel. In multitasking programming with several threads, you'll come across this deadlock problem often. This problem also is not unique to mutexes; it can also appear when using semaphores and critical sections. For the user this can result in an application that won't react to any input, and has, by all appearances, crashed. However, in reality its threads are still active, they're just stuck in their Wait calls, since they're effectively blocking one another.

However, deadlocks are not a problem that simply befalls an application without warning. They're normally the result of bad planning when considering how the program's threads work together. Deadlocks, therefore, usually appear in the developing and testing phase of an application. Fortunately, they can usually be solved fairly easily by analyzing the problem and modifying the affected program components.

Critical sections

It's possible to use a critical section wherever you might use a mutex. Critical sections are neither objects, nor referenced through handles, nor used with wait functions. However, they represent a very quick and efficient means of preventing a section of code from running with more than one thread at a time. While mutexes can also be used to synchronize threads in different processes, using critical sections is limited to the current process.

Critical sections are also called *code mutexes* since they serialize the access to a section of code. If a thread wishes to execute the code within a critical section while another is within the section, it will be blocked until the other thread has completed the critical section.

Use as many critical sections within a program as you like. You can, if necessary, protect code segments in completely different sections of the program with one single critical section. However, only one can run within this critical section at any one time. This approach thus only makes sense if the program components are related logically and use the same resources that should not be used by more than one thread at a time. However, if the different program sections use different resources, you should create a critical section for each of them.

Win32 critical section functions	
Function	Description
InitializeCriticalSection	Initializes a critical section object.
EnterCriticalSection	Acquires the thread that is making the function call for the critical section object
LeaveCriticalSection	Releases a critical section object.
DeleteCriticalSection	Deletes a critical section object.

Critical sections aren't created using an API function. They're instead implemented within an application as global variables of the CRITICAL_SECTION type. Before a thread enters the critical section for the first time, it must be initialized with InitializeCriticalSection(). This must only be done once for each critical section. We recommend doing it immediately at the start of the program so you don't have to deal with it later.

The appropriate function for this task is InitializeCriticalSection(). The only argument it requires is a pointer to the critical section variable that is to protect that particular critical section.

```
VOID InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
    // pointer to the critical section object
);
```

The critical section appears in the code through the EnterCriticalSection() and LeaveCriticalSection() functions. It starts with EnterCriticalSection() and ends with LeaveCriticalSection(). The code before and after the section can be used by any number of threads, but the code within the section can only be used by one thread at any given time. So, you might see EnterCriticalSection() as a type of gatekeeper who only permits one thread into the critical section. All others are blocked until this thread has left the critical section with the LeaveCriticalSection() function call.

```
VOID EnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
    // pointer to critical section object
);
VOID LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
    // pointer to critical section object
);
```

Both functions require a pointer to the CRITICAL_SECTION variable that is responsible for controlling that particular critical section. The following example shows that by using a critical section instead of a mutex the RpGetMessage() and RpPutMessage() functions can protect themselves against being run by more than one thread.

```
#include <windows.h>

//-- globale Variablen -----
```

```

CRITICAL_SECTION g_CritSecRpPut;           // critical section for RpPutMessage
CRITICAL_SECTION g_CritSecRpGet;           // critical section for RpGetMessage

typedef LONG MES;                           // ring buffer elements Elemente

#define RINGBUFFER_LEN 512                  // elements in the ring buffer

MES g_Ringbuffer[ RINGBUFFER_LEN ];         // the ring buffer
int g_iRpFirst, g_iRpLast;                  // current Put and Get index

//-- start function -----

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR     lpszCmdLine,
                   int        nCmdShow )
{
    //-- Critical Sections initialization
    InitializeCriticalSection( &g_CritSecRpPut );
    InitializeCriticalSection( &g_CritSecRpGet );

    g_iRpFirst = g_iRpLast = 0;              // ring buffer is empty at this point

    // .
    // threads are started that use RpPutMessage() to deposit message in the
    // ring buffer while others use RpGetMessage() to read them
    // via RpGetMessage() retrieve}}
    // .

    DeleteCriticalSection( &g_CritSecRpPut ); // delete the critical section
    DeleteCriticalSection( &g_CritSecRpGet );
    return 0;
}

//-- RpPutMessage() deposits a message in the ring buffer -----

BOOL RpPutMessage( MES m )
{
    BOOL bRetVal;
    EnterCriticalSection( &g_CritSecRpPut ); // entering the critical section

    if( g_iRpFirst != ( g_iRpLast + 1 ) % RINGBUFFER_LEN )
    { // there is still room in the ring buffer for more messages

        // deposit messages in the ring buffer and set g_iRpLast to next slot in
        // the ring buffer
        g_Ringbuffer[ g_iRpLast ] = m;
        if( ++g_iRpLast == RINGBUFFER_LEN ) g_iRpLast = 0;
        bRetVal = TRUE;
    }
    else bRetVal = FALSE;

    LeaveCriticalSection( &g_CritSecRpPut ); // leave the critical section
}

```

```

    return bRetVal;
}

//-- RpGetMessage() reads a message from the ring buffer -----

MES RpGetMessage( void )
{
    MES m;

    EnterCriticalSection( &g_CritSecRpGet );    // enter the critical section
    if( g_iRpFirst != g_iRpLast )
    {
        // at least one still unread message in the ring buffer
        m = g_Ringbuffer[ g_iRpFirst ];
        if( ++g_iRpFirst == RINGBUFFER_LEN ) g_iRpFirst = 0;
    }
    else
        // no message in the ring buffer, return -1
        m = -1L;
    LeaveCriticalSection( &g_CritSecRpGet );    // leave the critical section

    return m;
}

```

If you no longer need a critical section, use `DeleteCriticalSection()` to delete it. Then delete the accompanying `CRITICAL_SECTION` variable as well. A `EnterCriticalSection()` or `LeaveCriticalSection()` call is then no longer possible.

```

VOID DeleteCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
    // pointer to the critical section object
);

```

Critical sections aren't kernel objects. Instead they're formed by global variables. You also don't have to explicitly delete them at the end of a process with `CloseHandle()`. They simply expire similar to all the other global variables of a terminated process.

Semaphores

Semaphores are basically an expansion of the mutex idea. Although they also limit the access to a resource, they don't limit it to a single thread per access (unlike mutexes). The number of threads that can simultaneously access a resource through a semaphore is specified when the semaphore object is created. Consider, for example, a communication server within a network with five serial ports to which five modems are connected. These modems are protected with a semaphore so no more than five applications (threads) try to access them at any given time. The maximum number of possible threads would be set at five for this semaphore object.

Win32 semaphore functions	
Function	Description
CreateSemaphore	Creates a new semaphore or opens an existing one.
OpenSemaphore	Returns a handle to an existing semaphore.
ReleaseSemaphore	Releases a previously acquired semaphore.
Wait...	Threads can acquire a semaphore using the wait function.
CloseHandle	Deletes a semaphore object.

If fewer than five modems are in use, any thread that tries to access the semaphore will get the green light. However, if all the modems are in use, this attempt is blocked until at least one of the threads has released the semaphore. To keep track of the current number of users, semaphores use a form of internal user counter. If a thread tries to acquire a semaphore using

one of the different wait functions, the semaphore can check this counter to see whether it can grant the thread access to the resource protected by the semaphore. If not, the thread is blocked until otherwise notified.

CreateSemaphore() is used to create a semaphore and obtain its handle. Like mutexes, you can ignore the first and last parameters if the semaphore is to be used only within the current process. In this case, specify NULL for both.

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,    // Win95: NULL
    LONG lInitialCount,                             // initial counter
    LONG lMaximumCount,                             // maximum counter
    LPCTSTR lpName // pointer to string containing the
                  // name of the semaphore
);
```

The two parameters lInitialCount and lMaximumCount determine the initial and maximum values of the semaphore. This determines how many threads can acquire the semaphore at any given time (lMaximumCount) and how many will possess it immediately after it is created (lInitialCount).

The counter starts with the value in lInitialCount. Each time a caller acquires the semaphore using a wait function, this counter is decreased by 1. When it reaches 0, the maximum number of threads are using the semaphore. Additional threads will be blocked by their wait functions from acquiring the semaphore. The counter increased by 1 only if one of the threads in possession of the semaphore calls ReleaseSemaphore() to release it. If this raises the counter to a value above 0, the blocked thread can acquire the semaphore and continue its execution.

Remember the following regarding semaphores and wait functions: “A semaphore is signaled when the semaphore counter is greater than 0. It's nonsignaled when the counter has reached 0”. If, for example, a maximum of ten callers can acquire the semaphore at any one time, we must specify the value 10 for the lMaximumCount parameter with the CreateSemaphore() call. If the semaphore won't be acquired by any callers initially, we'll also need to set lInitialCount to 10. On the other hand, we can also set it to 0. This way no callers can acquire the semaphore until the handle returned by CreateSemaphore() is used with ReleaseSemaphore() to release the semaphore..

Semaphores are thus a generalized form of mutexes. In other words, mutexes are simply semaphores with a maximum value of 1. So, we could also refer to mutexes as binary semaphores. However, there is another small but significant difference between mutexes and semaphores. While mutexes can only be released by the thread that acquired it, any thread can release a semaphore using ReleaseSemaphore(). The program at the end of this chapter shows how this small difference was the deciding factor for using semaphores over mutexes.

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore, // handle of the semaphore object to be released
    LONG cReleaseCount, // increment semaphore counter by
    LPLONG lpPreviousCount // pointer to LONG, which accepts previous
                          // counter
);
```

The first argument required by ReleaseSemaphore() is the handle of the semaphore whose counter is supposed to be incremented. The second argument specifies by how much it is incremented. Although this value is usually 1, the semaphore are released multiple times in certain situations. Therefore, it makes sense to specify a higher value. What is important is that the semaphore counter doesn't exceed its maximum value specified with CreateSemaphore() when the semaphore was first created. This would result in an error. This error would be indicated in the function result that is returned, which would then be FALSE.

The third parameter allows you to obtain the previous value of the semaphore counter. The parameter must consist of a pointer to a long variable, to which the function will write previous value. If you don't need this information, simply specify NULL. This parameter also allows you to use ReleaseSemaphore() to simply check the status of the semaphore counter without incrementing it. Simply specify 0 for cReleaseCount so the counter value is not changed. The following partial program code shows how you can use this technique to make a simple macro for obtaining the current value of the semaphore counter:

```
#define GET_SEMAPHORE_COUNT(s,v) ReleaseSemaphore( s, 0, &(v) )

HANDLE h;
LONG CurCount;

h = CreateSemaphore( NULL, 10, 10, NULL);
GET_SEMAPHORE_COUNT( h, CurCount);
printf( Current Semaphore Counter: %ld\n", CurCount);
```

An existing semaphore can be opened at any time by another process using `OpenSemaphore()`. When and how to use this function is explained in the “Synchronization between processes” section.

```
HANDLE OpenSemaphore(
    DWORD      fdwAccess,           // desired access to the semaphore object
    BOOL       fInherit,           // TRUE: the semaphore's handle can be inherited
    LPCTSTR    lpzszSemName       // pointer to a string with the semaphore's name
);
```

As with mutexes, `CloseHandle()` is used to delete semaphore objects before a process is deleted.

```
BOOL CloseHandle(
    HANDLE hObject                // semaphore's handle
);
```

Semaphores for controlling overflow

Let's return to the ring buffer example one more time. Semaphores can also be quite helpful in implementing the `RpPutMessage()` and `RpGetMessage()` functions, although in a very different manner than you might expect. Although it would be possible to replace the mutexes with semaphores whose maximum value is set at 1, this is not the point we're trying to get at. Rather, the use of semaphores offers a very practical opportunity to pause the execution of `RpPutMessage()` and `RpGetMessage()` when

- The ring buffer is already full when `RpPutMessage()` is called, or
- It is empty when `RpGetMessage()` is called.

In the previous example we weren't particularly concerned with these two scenarios, since the focus was preventing the simultaneous execution of the functions by two or more functions. However, this is not a problem you can ignore when you're writing a real application. It's possible the buffer is full or empty at the worst possible time.

These conditions within the functions are easily detected with IF statements. This makes it possible to respond to the situation with an appropriate error code. However, if we want to pause the thread until there is either more room in the ring buffer (`RpPutMessage()`) or until it contains at least one message (`RpGetMessage()`), semaphores offer a good solution.

The following listing shows this solution. Besides the two critical sections, which are still needed, we've also created one semaphore for `RpPutMessage()` and one for `RpGetMessage()`. The semaphore for `RpPutMessage()` is named `g_hSemRpPut`, the other `g_hSemRpGet`. The internal counter of `g_hSemRpPut` is to represent the number of slots (array elements) in the ring buffer that are still free, and is therefore initialized with the length of the ring buffer. The internal counter of `g_hSemRpGet`, on the other hand, will represent the number of messages in the ring buffer that haven't yet been read. It is therefore initialized with 0.

```
#include <windows.h>

//— types and constants —————
```

```

typedef LONG MES;                                     // ring-buffer elements
#define RINGBUFFER_LEN 512                             // ring buffer size

//— global variables —————

CRITICAL_SECTION g_CritSecRpPut;                       // critical section for RpPutMessage()
CRITICAL_SECTION g_CritSecRpGet;                       // critical section for RpGetMessage()

HANDLE g_hSemRpPut;                                    // semaphore for RpPutMessage()
HANDLE g_hSemRpGet;                                    // semaphore for RpGetMessage()

LONG lTimeOut = 10000L;                                // timeout after 10 seconds

MES g_Ringbuffer[ RINGBUFFER_LEN ]                    ; // the actual ring buffer
int g_iRpFirst, g_iRpLast;                             // index for Put and Get

//— start function —————

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR     lpszCmdLine,
                   int       nCmdShow )
{ int i;
  // create critical sections for Put and Get
  InitializeCriticalSection( &g_CritSecRpPut );
  InitializeCriticalSection( &g_CritSecRpGet );

  // create semaphore for Put, set to ring buffer length
  g_hSemRpPut = CreateSemaphore( NULL, RINGBUFFER_LEN, RINGBUFFER_LEN, NULL );

  // create semaphore for Get, set to 0
  g_hSemRpGet = CreateSemaphore( NULL, 0, RINGBUFFER_LEN, NULL );

  g_iRpFirst = g_iRpLast = 0;                          // set both indices to first slot

  // .
  // threads are started that use RpPutMessage() to deposit messages in the
  // ring buffer, while others use RpGetMessage() to read these messages
  // .

  CloseHandle( g_hSemRpPut );                          // delete the two semaphores again
  CloseHandle( g_hSemRpGet );

  DeleteCriticalSection( &g_CritSecRpPut );            // release the two
  DeleteCriticalSection( &g_CritSecRpGet );            // critical sections again

  return 0;
}

//— RpPutMessage() deposits a message in the ring buffer —————
void RpPutMessage( MES m )
{
  // wait for free slot (Put-semaphore not equal to 0)

```



```

    WaitForSingleObject( g_hSemRpPut, INFINITE);

    // free slot available, enter the critical section
    EnterCriticalSection( &g_CritSecRpPut );

    // deposit a new message in the ring buffer
    g_Ringbuffer[ g_iRpLast ] = m;
    if( ++g_iRpLast == RINGBUFFER_LEN ) g_iRpLast = 0;

    // increment Get-semaphore (one more slot taken)
    ReleaseSemaphore( g_hSemRpGet, 1, NULL );

    // leave critical section again
    LeaveCriticalSection( &g_CritSecRpPut );
}

//— RpGetMessage() reads a message from the ring buffer ———
MES RpGetMessage( void )
{
    MES m;

    // wait until there is at least one message in the ring buffer
    // (Get-semaphore not equal to 0)
    WaitForSingleObject( g_hSemRpGet, INFINITE);

    // message in ring buffer, enter the critical section
    EnterCriticalSection( &g_CritSecRpGet );

    // read message from the ring buffer
    m = g_Ringbuffer[ g_iRpFirst ];
    if( ++g_iRpFirst == RINGBUFFER_LEN ) g_iRpFirst = 0;

    // increment Put-semaphore (one less slot taken)
    ReleaseSemaphore( g_hSemRpPut, 1, NULL );

    // leave critical section again
    LeaveCriticalSection( &g_CritSecRpGet );

    return m;
}

```

What this is all getting at becomes more clear when we take a look at RpPutMessage() and RpGetMessage(). The first statement in RpPutMessage() is now:

```

RpPutMessage():

    // wait for free slot (Put-semaphore not equal to 0)
    WaitForSingleObject( g_hSemRpPut, INFINITE);

```

The caller can only run the function if g_hSemRpPut is not equal to 0. In other words, it runs the function only if there is at least one more available slot. When the wait function returns to the caller, the internal counter has already been decreased and thus correctly indicates that there is now one fewer available slot. After all, a message is to be placed in the ring buffer.

By the same token, the semaphore counter of `g_hSemRpPut` has to be incremented when a message is read from the ring buffer. This occurs in the second to last line of `RpGetMessage()`:

```
RpGetMessage():

    // increment Put-semaphore (one less slot taken )
    ReleaseSemaphore( g_hSemRpPut, 1, NULL );
```

The opposite is done with the semaphore `hSemRpGet`. Here it's the `RpGetMessage()` function that right at the start tries to acquire the semaphore.

```
RpGetMessage():

    // wait until there is at least one message in the ring buffer
    ..// (Get-semaphore not equal to 0)
    WaitForSingleObject( g_hSemRpGet, INFINITE);
```

If the internal semaphore counter of `hSemRpGet` is at 0, because the ring buffer is empty, the thread that made the function call is blocked until a new message arrives in the ring buffer. The semaphore is therefore released at the end of `RpGetMessage()` so the internal semaphore counter is incremented after a new message has been written to the ring buffer.

```
RpPutMessage():

    // increment Get-semaphore (one more taken slot)
    ReleaseSemaphore( g_hSemRpGet, 1, NULL );
```

If no message was in the ring buffer before, a thread that has been blocked by the wait call at the start of `RpGetMessage()` is thereby continued. The message that was just written to the ring buffer is returned. Therefore, the point at which the `ReleaseSemaphore()` call is made is also very important. This function must only be called after the message has been written to the buffer. Otherwise, a thread that might be waiting in `RpGetMessage()` would be continued and then attempt to read the message, although it's not yet in the ring buffer. The same is true for `RpPutMessage()`. First, it's necessary to create a free slot through the wait function call before the message may be written to the ring buffer.

By looking at the two functions, you'll see that we're still using a critical section despite also using semaphores. This is to make certain that only one caller can access the function at any time. After all, the semaphores are only used here to prevent the overflow or underflow of the ring buffer. They have nothing to do with the simultaneous use of the function by multiple function callers.

If you've never implemented this type of solution, it may seem somewhat complicated at first. However, if you've familiarized yourself with using synchronization objects, you'll most likely find the solution rather efficient. After all, it comes down to less than four lines of code that prevent the ring buffer's overflow and underflow. Also, these lines minimize the workload for the processor since they don't force the caller to keep calling the functions repeatedly if the ring buffer is full or empty.

Event objects

Events fall into an entirely different category than mutexes, semaphores and critical sections. The point with events is not to coordinate access to a resource but rather to stop the execution of threads when certain events happen. This allows a thread to control the execution of another, for example that of a background thread which is supposed to begin calculating data only once the foreground thread has finished loading them into memory.

In this case, the foreground thread would first create an event object using `CreateEvent()` and store the returned handle in a global variable. It would then start the background thread. The latter would wait for the event by specifying the events handle with a wait function call. The function will only return once the foreground thread has used `SetEvent()` or `PulseEvent()` to signal the event. For the background thread, this is the signal to begin processing the data. However, if the event hasn't been signaled, the wait function prevents the thread from running.

We could also use a global variable to achieve the same end. The background thread would simply check this variable (let's call it GO) in an endless loop until the foreground thread sets the variable to TRUE. Only then would the background thread begin processing the data. However, this approach has an important disadvantage: it wastes processor time needlessly. It keeps receiving processor time while the background thread is waiting for the foreground thread. This processor time is wasted completely because all the background thread does is check the GO variable repeatedly.

```
#include <windows.h>

//-- global variables -----

BOOL bGo;          // indicates to background thread whether it may proceed

//-- the background thread -----

DWORD WINAPI Calculate( LPVOID lpParms )
{
    while( !bGo )          // run loop until GO is set to TRUE
        ;

    // .
    // Here we go!!!
    // .

    return 0;
}

//-- start function -----

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR      lpszCmdLine,
                   int        nCmdShow )
{
    HANDLE hThread;
    DWORD  dwThreadId;

    bGo = FALSE;          // prevent premature start of background thread

    hThread = CreateThread( NULL, 0L, Calculate, NULL, 0L, &dwThreadId );

    // .
    // read data that will be processed by background thread
    // .

    bGo = TRUE;  // let background thread start processing the data

    // .
    // .
    // .

    WaitForSingleObject( hThread, INFINITE );          // wait for thread to end
    CloseHandle( hThread );                          // delete thread object
}
```

```
    return 0;
}
```

However, by using an event object, the background thread receives no processor time whatsoever after the wait function call is made. Only the foreground thread and other nonblocked threads are executed until the event object is signaled. The system recognizes that the background thread is waiting for this event, and therefore releases it for scheduling again. Thus no processor time is wasted within by the background thread.

```
#include <windows.h>

//— global variables —————

HANDLE g_hEventGo;    // Handle to Go-Event

//— the background thread —————

DWORD WINAPI Berechne( LPVOID lpParms )
{
    // wait for go-event to be signaled

    WaitForSingleObject( g_hEventGo, INFINITE );

    // .
    // off we go...
    // .

    return 0;
}

//— start function —————

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR      lpszCmdLine,
                   int         nCmdShow )
{
    HANDLE hThread;
    DWORD  dwThreadId;

    // create go-event as a nonsignalized auto-reset event
    g_hEventGo = CreateEvent( NULL,
                             FALSE,                          // auto-reset event
                             FALSE,                          // not signalized
                             NULL );

    hThread = CreateThread( NULL, 0L, Berechne, NULL, 0L, &dwThreadId );

    // .
    // read the data that is to be processed by the background thread
    // .
}
```

```

// signalize event ù this starts the background thread
SetEvent( g_hEventGo );

//...

WaitForSingleObject( hThread, INFINITE );// wait for the end of the thread

CloseHandle( hThread );                                // delete objects
CloseHandle( g_hEventGo );
return 0;
}

```

The following table summarizes the functions that Win32 API provides for working with events:

Win32 event functions	
Function	Description
CreateEvent	Creates a new event object or opens an existing one.
OpenEvent	Returns a handle to an existing event object.
SetEvent	Places an event object in the signaled state.
ResetEvent	Places an event object in the nonsignaled state.
PulseEvent	Triggers an event object.
Wait...	Thread uses the Wait function to wait for the event to be signaled.
CloseHandle	Deletes an event object.

Event objects are created by the CreateEvent() function. If you only need the event object to synchronize threads within the current process, you can specify NULL for the first and last function parameters.

```

HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,          // Win95: NULL
    BOOL bManualReset,      // auto reset or manual reset?
    BOOL bInitialState,     // initial status of the event
    LPCTSTR lpName          // pointer to a string containing the
    .....// event's name
);

```

The second parameter of CreateEvent(), bManualReset, determines which type of event object will be created. The Win32 API distinguishes between two different types of events: auto reset events (bManualReset = FALSE) and manual reset events (bManualReset = TRUE). The difference between these two types of events becomes evident when a thread uses SetEvent() to place a nonsignaled event object into the signaled state. A manual reset event will then remain in the signaled state, and all threads that have been waiting for the event through a Wait function will run again. Only an explicit ResetEvent() call will place the event back into the nonsignaled state.

The following is different regarding auto reset events: When a thread that has been waiting for an event with Wait...() begins running again once the event has been signaled, the event object is automatically placed back into the nonsignaled state. If several threads are waiting for the event, it must be signaled repeatedly using SetEvent(), so the threads start running again one by one. However, if no thread is waiting for the event object, it will remain signaled until a thread makes a wait function call on the object.

The bInitialState parameter with the CreateEvent() function call determines whether a newly created event object will appear in a signaled or nonsignaled state. If it's initialized with TRUE, the object will be signaled; any other value will create a nonsignaled event object.

Use `OpenEvent()` to obtain a handle to an existing event object. This handle can then be used to use the other event functions to work with that object. For more information see the section on Synchronization between processes.

```
HANDLE OpenEvent(
    DWORD    fdwAccess,           // desired access to the event object
    BOOL     fInherit,           // TRUE, if the object's handle can be inherited
    LPCTSTR  lpszEventName       // pointer to a string containing the
                                // event's name
);
```

An event object is switched between the signaled and nonsignaled states by the functions `SetEvent()` and `ResetEvent()`. `SetEvent()` places the object with the specified handle in the signaled state, `ResetEvent()` places it in the nonsignaled state. Both functions return the value `TRUE` when they are executed successfully, in other words, when the object was placed in the desired state. In the case of an error (return value = `FALSE`), you can use `GetLastError()` to obtain more detailed error information. The most common error is an incorrect handle.

```
BOOL SetEvent(
    HANDLE hEvent                // handle of the event object
);
BOOL ResetEvent(
    HANDLE hEvent                // handle of the event object
);
```

`PulseEvent()` combines `SetEvent()` and `ResetEvent()`. `PulseEvent()` places the event object in the signaled state and then immediately switches it back to its nonsignaled state. With a manual reset event, all the threads that have been waiting for the event through a wait function will thereby be reactivated before the event is reset again. With an auto reset event, on the other hand, only the one of the waiting threads will be continued before the event object is placed back in the nonsignaled state.

```
BOOL PulseEvent(
    HANDLE hEvent                // handle of the event object to be pulsed
);
```

As with mutexes and semaphores, `CloseHandle()` is used to delete an event object. The function argument requires the handle of the event object that you want to close. The function returns the value `TRUE` after successfully deleting the object.

```
BOOL CloseHandle(
    HANDLE hObject               // handle of the event object
);
```

Interlocked variables

Interlocked variables provide a basic method for manipulating `LONG` variables when the latter are used simultaneously by multiple threads to control the course of the program. Multitasking can pull the carpet out from under you when you use this type of variable, first incrementing it and then comparing its value; namely when a switch to another thread occurs precisely at the moment between the incrementation and when the variable's value is checked. If the new thread also modifies the variable's value and the old thread is continued shortly thereafter, the latter will continue by checking the variable's value. However, since the other thread has changed this value, the check will be faulty.

The incrementing and checking of the variable must therefore be joined into a unit that will not be interrupted by a switch in tasks. The Win32 API has three functions for this purpose so we have an alternative to always enclosing such a code sequence within a critical section. Similar functions are used within the kernel for the management of mutexes and semaphores.

Win32 functions for interlocked variables	
Function	Description
InterlockedDecrement	Decrements (decreases) a DWORD that is shared with other threads.
InterlockedExchange	Assigns a new value to a DWORD that is shared with other threads.
InterlockedIncrement	Increments (increases) a DWORD that is shared with other threads.

InterlockedIncrement() and InterlockedDecrement() are used to increment or decrement a long variable and then compare it with 0. Both functions require a pointer to the desired variable. The returned value indicates whether the interlocked variable is now equal to, greater than or less than zero. In the first instance, the returned value is 0. If the variable has assumed a value greater than zero, a value greater than zero is returned. A value smaller than zero is returned if the variable has assumed a negative value.

```

LONG InterlockedDecrement(
    LPLONG lpAddend    // pointer to the DWORD that is to be decremented
);
LONG InterlockedIncrement(
    LPLONG lpAddend    // pointer to the DWORD that is to be incremented
);

```

InterlockedExchange() is used to assign a new value to an interlocked rather than incrementing or decrementing its previous value. Besides the mandatory pointer to the variable, this function also requires the new value. It returns the value previously contained in the variable.

```

LONG InterlockedExchange(
    LPLONG Target,      // pointer to the DWORD that is to be loaded
    LONG Value          // new value for the DWORD
);

```

Processes and threads as synchronization objects

Processes and threads can also be used as synchronization objects with a wait function. This is especially the case when a thread is supposed to wait for the termination of a process or thread before continuing its execution. In this sense, a process or thread can function like an event bearing the message “I am now ready.”

Processes and threads are considered signaled when they’re terminated and nonsignaled if they’re running (or they’re waiting for a synchronization object and are therefore blocked). The following call:

```
WaitForSingleObject( hThread, INFINITE );
```

pauses the current thread until the thread identified by hThread has been terminated. In the same manner, it’s possible to use WaitForSingleObject() to specify a process handle. You’ll find more information on the second parameter, INFINITE, later.

Synchronization functions

In the preceding sections we frequently talked about wait functions although we’ve really only introduced WaitForSingleObject(). This function pauses the thread that made the function call until the synchronization object with the specified handle enters the signaled state. However, WaitForSingleObject() is only one of several wait functions with which we can use the available synchronization objects. Before looking at these functions, the following table summarizes the different synchronization objects that can be used with the wait functions:

Object	Description
Change-Notification	Becomes signaled when a change is made within a directory or a directory tree.
Console input	Becomes signaled when a yet unprocessed character enters the input buffer.
Event	Becomes signaled when SetEvent() or PulseEvent() is called.
Mutex	Is signaled if the mutex is not acquired by another thread or its owner releases it with ReleaseMutex().
Semaphore	Is signaled if the semaphore counter is not equal to 0, in other words, if the semaphore can still be acquired by further callers.
Process	Becomes signaled when the process is terminated.
Thread	Becomes signaled when the thread is terminated.

We have to distinguish between two types of wait functions:

1. Those that only wait for one synchronization object.
2. Those that wait for more than one synchronization object.

The Win32 API provides two functions for each of these. A standard version and an expanded version that is used with asynchronous I/O accesses.

Win32 synchronization functions	
Function	Description
WaitForSingleObject	Waits until a specified object enters the signaled state or a timeout occurs.
WaitForSingleObjectEx	Like WaitForSingleObject(), however, it also reacts to the completion of an asynchronous I/O operation.
WaitForMultipleObjects	Pauses the execution of the current thread until one or more synchronization objects have entered the signaled state.
WaitForMultipleObjectsEx	Like WaitForMultipleObjects(), however, it also reacts to the completion of an asynchronous I/O operation.
MsgWaitForMultipleObjects	Waits for the availability of one or more synchronization objects, for input, or for a timeout to expire.

WaitForSingleObject() is the simplest type of wait function. It blocks the thread that has made the function call until the specified synchronization object is signaled or until a specified amount of time has passed (timeout). If the specified object is already signaled when WaitForSingleObject() is called, the function returns to the caller immediately so the thread can continue running.

```

DWORD WaitForSingleObject(
    HANDLE hObject,                // handle of the object
    DWORD dwTimeout                // maximum wait time in milliseconds
);

```

The second parameter, dwTimeout, allows you to specify the maximum number of milliseconds the function will wait. If the specified synchronization object is not signaled within this amount of time, WaitForSingleObject() will return to the caller regardless. If the function is to wait for the specified synchronization object with no time limit, specify INFINITE for this parameter.

It's useful to specify a maximum wait period with a timeout value whenever you don't want the program to let the user wait too long. Let's use the example with the semaphores from above that were used to control the access to several modems in a network. Depending on what these modems might be used for at any given time, it might require some time before one of them is available again. Limit the timeout of the wait function to a few seconds. Then the user won't have to wait too long and perhaps receives a message from the application. If the application was unable to acquire the modem (i.e. the semaphore)

within this time, this will allow us to at least display a message to the user. Then we can try again to acquire the modem, until the user either aborts the attempt, or until the application quits trying after a predetermined number of failed attempts.

The function result of `WaitForSingleObject()` indicates whether the desired synchronization object was able to be signaled. If `WAIT_OBJECT_0` is returned, the signalization has succeeded, `WAIT_OBJECT_0 + WAIT_TIMEOUT` means the timeout has been reached instead.

If you're waiting for a mutex object, the function may also return another function result: the constant `WAIT_ABANDONED`. It indicates the thread that possessed the mutex was terminated without first releasing the mutex. Although this will let the mutex be acquired by the new thread, it still indicates that an error has most likely occurred in the program. After all, threads should always explicitly release mutexes before they are terminated. If this doesn't happen, then an error may have occurred through which the thread was terminated prematurely.

```
DWORD WaitForSingleObjectEx(
    HANDLE hObject,                                // handle of the object
    DWORD dwTimeout,                               // maximum wait time in milliseconds
    BOOL fAlertable                                // TRUE: return when asynchronous
                                                // I/O operation is completed })
);
```

`WaitForSingleObjectEx()` works the same as `WaitForSingleObject()` except that it requires another parameter, `fAlertable`. `WaitForSingleObjectEx()` is intended for threads that initiate an asynchronous I/O operation with `ReadFileEx()` or `WriteFileEx()` before calling the wait function. Generally such a thread will want to be notified of the completion of the I/O operation, even if the specified synchronization object hasn't yet been signaled. Therefore, if the thread specifies `TRUE` for the `fAlertable` parameter with the `WaitForSingleObjectEx()` function call, the function will return to the thread at the latest when the I/O operation is completed, regardless of the synchronization object's status. Besides the different return values of `WaitForSingleObject()`, `WaitForSingleObjectEx()` also includes the return value `WAIT_IO_COMPLETION`. This return value indicates that the asynchronous I/O function has been completed.

If a thread can use `WaitForMultipleObjects()` or `WaitForMultipleObjectsEx()` to make its execution dependent multiple synchronization objects, rather than just one. These functions are useful in situations where a thread needs to wait for the termination of more than one thread. It will be continued only once all of them have ended. The two functions are identical except for the parameter `fAlertable` to accommodate asynchronous I/O operations with `WaitForMultipleObjectsEx()`.

```
DWORD WaitForMultipleObjects(
    DWORD cObjects,                                // number of handles in the array
    CONST HANDLE *lphObjects,                      // pointer to the array with the handles
    BOOL fWaitAll,                                 // TRUE: wait until all objects are signaled
    DWORD dwTimeout                                // maximum wait time in milliseconds
);
DWORD WaitForMultipleObjectsEx(
    DWORD cObjects,                                // number of handles in the array
    CONST HANDLE *lphObjects,                      // pointer to the array with the handles
    BOOL fWaitAll,                                 // TRUE: wait until all objects are signaled
    DWORD dwTimeout,                               // maximum wait time in milliseconds
    BOOL fAlertable                                // TRUE: return when asynchronous I/O operation
                                                // is completed
);
```

The first parameter, `cObjects`, specifies the number of synchronization objects for which the function will wait. The handles of these objects are referenced through an array. The thread that makes the function call must first create this array and load it with the handles of the desired objects. The parameter `lphObjects` specifies a pointer to this array. The `fWaitAll` parameter determines whether the function will only return to the caller when each of the specified handles has entered the signaled

state or once just one of these has been signaled. If you specify TRUE for this parameter, all the objects will have to be signaled — this essentially results in a type of AND operator. Any other value represents an OR operator. In the latter case, the function will return when any one of the specified synchronization objects is signaled.

One characteristic regarding mutexes and the AND operator you must remember: The specified mutexes are acquired by the thread that made the function call only once all other synchronization objects have been signaled. If another thread runs a wait function on one of these mutexes in the meantime, it can acquire it. In that case, the first thread will have to wait until the other one releases the mutex again. This is true even if all its other synchronization objects have already been signaled.

As with `WaitForSingleObject()` and `WaitForSingleObjectEx()`, the fourth parameter represents a timeout value in milliseconds. Again, you can specify INFINITE if you don't want to impose a time limit on the function.

The function results of the two `WaitForMultipleObject()` functions are a little more complex to evaluate than those of the `WaitForSingleObject()` functions. If you specify an AND operator (`fWaitAll = TRUE`), you'll get the same function results as with the single object functions:

WAIT_OBJECT_0	If all the desired synchronization objects have been signaled.
WAIT_ABANDONED_0	If all the desired synchronization objects have been signaled, although at least one of the specified mutexes was signaled in that its thread was terminated.
WAIT_TIMEOUT	If none of the synchronization objects were acquired within the specified amount of time. In this case, requested mutexes will not be acquired, even if this would otherwise be possible.

However, if you've used an OR operator (`fWaitAll = FALSE`), the function result indicates the number of the element within the handle array that has been signaled and which has therefore triggered the function return to the thread that made the function call. Assuming the function wasn't ended by a timeout (`fWaitAll = FALSE`), you'll usually get a value between `WAIT_OBJECT_0` and `(WAIT_OBJECT_0 + cObjects - 1)`. Subtract the constant `WAIT_OBJECT_0` from this value to determine the index number of the signaled synchronization object within the array. This tells you which of the specified synchronization objects has caused the function to return to the thread.

A function result with a value between `WAIT_ABANDONED_0` and `(WAIT_ABANDONED_0 + cObjects - 1)` occurs if the synchronization object that has been signaled is a mutex that has been signaled because its previous owner was terminated. Again, subtract `WAIT_ABANDONED_0` to get the index number of the mutex within the specified handle array.

```

DWORD MsgWaitForMultipleObjects(
    DWORD      nCount,           // number of the handle within the array
    LPHANDLE   pHandles,        // pointer to the array containing the handles
    BOOL       fWaitAll,         // TRUE: wait until all objects of the array
                                // have been signaled
    DWORD      dwTimeOut,        // maximum wait time in milliseconds
    DWORD      dwWakeMask,       // messages in the queue
    ...        // that trigger the return of the function
);

```

`MsgWaitForMultipleObjects()` is a special version of `WaitForMultipleObjects()`. Its syntax is identical to `WaitForMultipleObjects()` except for the parameter `dwWakeMask`. While the return of `WaitForMultipleObjectsEx()` is triggered by asynchronous I/O operations, `MsgWaitForMultipleObjects()` reacts to messages in the thread's queue that haven't yet been processed. It, therefore makes sense to use `MsgWaitForMultipleObjects()` wherever you need to wait for the signalization of one or more synchronization objects but don't want to lose sight of the interaction with the user.

The parameter `dwWakeMask` determines which messages will trigger the return of the function. You can specify any desired combination of the following constants:


```

LONG                lInitialCount,
LONG                lMaximumCount,
LPCTSTR    lpName                // allows a system wide name
                                // to be assigned
);

```

Shared use through an inherited handle

Like `CreateProcess()` and `CreateThread()`, all three `Create...` functions require a security descriptor for their first parameter. If a synchronization object will only be used within one process, specify `NULL` for this parameter. However, if the returned handle will be inherited by a child process, the security descriptor must not be neglected. It contains an inheritance flag which determines whether the newly created synchronization object can be inherited. If you specify `NULL` for the required pointer to the security descriptor, this flag will be set to `FALSE` so the handle will not be able to be inherited.

If you wish to prevent this, create a security descriptor of the `SECURITY_ATTRIBUTES` type, as defined in `WINBASE.H`, before you make the `Create...` function call.

```

typedef struct _SECURITY_ATTRIBUTES {
    DWORD    nLength;                // length of the structure
    LPVOID    lpSecurityDescriptor; // the actual security descriptor
    BOOL    bInheritHandle;         // can the handle be inherited?
} SECURITY_ATTRIBUTES;

```

The `nLength` field must then be initialized with the length of the structure (in other words, with `sizeof(SECURITY_ATTRIBUTES)`). You can load the field with the value `NULL` since this is the actual security descriptor, which is not needed here. Finally, `bInheritHandle` needs to be initialized with `TRUE` so the handle to the new synchronization object can be inherited. However, note that this measure only allows the handle to be inherited. How the handle gets from the parent process to the child process is another matter. Also see the comments on inheritance with `CreateProcess()` concerning this issue.

Shared use through a name

Another method of using a synchronization object in more than one process is through a system wide name. It's created by the argument `lpName`, which is the last parameter in the three `Create...` functions. If you specify a pointer to a string for this parameter, rather than `NULL`, the new synchronization object will receive the name specified in this string. This name can contain all characters except for the backslash (`\`). It can have a maximum of 64 characters.

The shared use of the synchronization object begins when another process calls a `Create...` function with the same name. Be aware that the name is case-sensitive. If the specified synchronization object already exists, the function doesn't create a new one, but instead returns to the caller a handle to the existing object. With this handle and the available synchronization functions, such as `WaitForSingleObject()`, the two processes can then be synchronized. Other processes can also acquire the synchronization object in the same way, making it possible to synchronize virtually any desired number of processes.

What is important is that you select a unique name for the synchronization object. This prevents creating conflicts with other processes with which you don't intend to work. Don't select names like "A", "B" or "C." Instead, use more complex constructs like "MyMarvellousMutex", "PrintFileEvent01", etc. Also make certain the names are unique between the various groups of synchronization objects. An error will result if you attempt to create a semaphore with a name that is already being used for a mutex or another synchronization object.

Opening an existing synchronization object

However, the three `Create...` functions don't represent the only way to obtain the handle of an existing synchronization object. Besides a `Create...` function, the Win32 API also provides an `Open...` function for mutexes, semaphores, and events, called `OpenMutex()`, `OpenSemaphore()`, and `OpenEvent()`. With these functions, a process can obtain a handle to a synchronization object that was previously created through the appropriate `Create...` function. All three functions use the same syntax:

```

HANDLE OpenMutex(
    DWORD    fdwAccess,           // desired access to the mutex object
                                   // MUTEX_ALL_ACCESS = all access rights
    BOOL     fInherit,           // TRUE if the object's handle can be inherited
    LPCTSTR  lpszMutexName       // pointer to a string containing the mutex
                                   // object's name
);

HANDLE OpenSemaphore(
    DWORD    fdwAccess,           // desired access to the semaphore object
                                   // SEMAPHORE_ALL_ACCESS = all access rights
    BOOL     fInherit,           // TRUE: object handle can be inherited
    LPCTSTR  lpszSemName         // pointer to a string with the name of the
                                   // semaphore object
);

HANDLE OpenEvent(
    DWORD    fdwAccess,           // desired access to the event object
                                   // EVENT_ALL_ACCESS = all access rights
    BOOL     fInherit,           // TRUE if the object's handle can be inherited
    LPCTSTR  lpszEventName       // pointer to a string with the name of the event
                                   // object
);

```

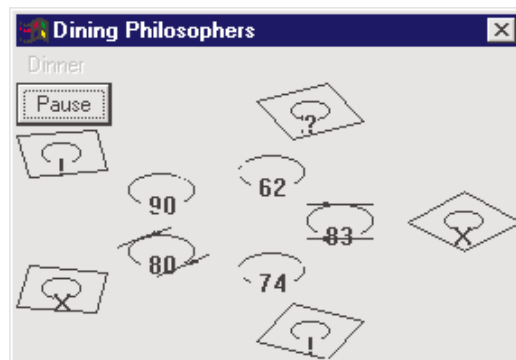
The first parameter determines the access rights to the handle that is returned. Specify the constant `MUTEX_ALL_ACCESS`, `SEMAPHORE_ALL_ACCESS` or `EVENT_ALL_ACCESS`, depending on the type of object, to allow the handle to be used with all the synchronization functions for the object type. The second parameter is `fInherit`. It determines whether the returned handle can be inherited by a child-process. Specify `TRUE` for this parameter if the handle is to be inherited. The last parameter identifies the synchronization object that you wish to acquire. If the name you specified can't be found, the function will return the value `NULL`. Otherwise, the handle to the desired synchronization object is returned. Don't forget to delete handles that you've acquired in this manner with `CloseHandle()` before the process is ended.

Sample Programs: Synchronization Between Philosophers

In this section we'll use a popular philosophical example to illustrate the extensive use of synchronization objects. This problem and its solution were originally described by E.W. Dijkstra in the 1960s in *The Dining Philosophers*. The problem consists of a simulation whose setting is the cafeteria of an imaginary university.

The professors of the philosophy department have taken seats at a round table to share their midday meal of spaghetti. The philosophy professors are excited about the spaghetti meal. However, they've noticed one problem: each philosopher has only one fork next to their plate. The philosophers will have a difficult time using only one fork to eat the spaghetti. Although a second fork is needed, the nearest one belongs to a neighbor. So, our professors reach for their neighbor's fork to their right

Philosophers at the table



or left. Those who are fast enough manage to borrow the essential second fork. The slower philosophers, however, suddenly find themselves without a fork and will remain hungry for awhile. Philosophy professors, being cultivated individuals, won't stoop so low as to eat spaghetti with their hands or start a fight with a colleague over a fork.

The slower professors therefore have no option but to hope their quicker neighbors will eventually put down their forks. When

a professor places a fork back, their neighbor will immediately reach for the fork so they can begin eating. You've probably determined the result. A constant struggle between professors who are eating their spaghetti with the neighbor's fork and others who are waiting to grab the fork it's placed back on the table. The DINNER.C program illustrates this academic dining chaos. It displays the sequence on the screen with a number of professors that can vary from 2 to 20.

Each of the professors starts with 100 pieces of spaghetti, which are eaten one after the other. The professors alternate among the following states (this continues until they are finished):

- Eating, when the professor has two forks in hand and is eating spaghetti.
- Thinking, when the professor's philosophical nature moves him or her to put the fork down and temporarily become engrossed in esoteric conversation.
- Hungry, when the professor again remembers the spaghetti and hopes that one of the neighboring philosophers will soon put down their fork so he or she can continue eating.

These different states are indicated on the screen by symbols that appear over each of the professors. An exclamation mark represents a hungry professor. A question mark represents a thinking professor. An X represents an eating professor. You can also see the two forks at the plate of an eating professor and the number of remaining pieces of spaghetti.

Before continuing with the program code, take some time to experiment with the program. You'll find it on the companion CD-ROM as DINNER.EXE in the \WIN95\DINNER\BIN directory. Use the Dinner menu to select the number of professors (2-20) at the lunch session. Use the Pause button in the upper lefthand corner if the action is too fast to pause the program. Select the Pause button again to continue the chaotic lunch.

Internal realization of DINNER

The project file includes the files DINNER, DINNER.RC, RESOURCE.H, TURTLE32.C and TURTLE32.H. The actual program is found in DINNER.C.

The constants for the first three professor conditions, THINKING, HUNGRY and EATING are defined at the start of DINNER.C. Also, two macros (LEFT_NEIGHBOR and RIGHT_NEIGHBOR) are defined. These provide the numbers of the left and right neighbor, respectively, for any given professor. The two macros are based on the global variable iNumPhilosophers. This global variable stores the number of professors that was specified by the user. The current state that each philosopher is in is stored in the array nSpaghetis. The number of remaining spaghetti pieces is stored in the array nSpaghetis. The philosophers are drawn using TURTLE32.C. The global variable TC receives the current turtle context. The philosopher window is handled as a dialog box, whose handle is stored in the global variable hMainDialog.

Each of the individual philosophers is created by a single thread. The thread handles are stored in the global array hThreadPhilosophers. Another array called hSemPhilosophists controls the access to the forks. This array contains a handle for each philosopher to a semaphore. The counter of this semaphore toggles between the values 0 and 1, depending on the state of its professor. A philosopher who isn't eating is signaled (semaphore-counter = 1). A professor who begins eating is entered in the signaled state (counter = 0). This idea will become clearer in following paragraphs.

While they are running, the philosopher threads change the philosophers' states by manipulating the nState array. A mutex is introduced to avoid race conditions during write access to this array. Its handle is stored in the global variable hMutexStateChange. Then each of the threads can access it.

bStopped and hMutexStartStop are used to control the program's execution with the Pause button in the upper lefthand corner of the window. bStopped always contains TRUE when the user stops the philosophers by clicking the Pause button. hMutexStartStop receives the handle of a mutex through which the bunch of hungry professors can be paused.

Note that the dialog box and not a window is set up in the application's WinMain() start function. It's entered into the normal message queue through which the messages can reach the DialogProc() dialog box.

When the WM_INITDIALOG message is received, the mutex for pausing the professors is first created. Its handle is stored in the global variable hMutexStartStop. Furthermore, the Turtle module is initialized, so the application can use it. When WM_PAINT is received, each philosopher is redrawn with DrawPhilosoph().

Action becomes more exciting when the dialog function receives a WM_COMMAND message after one of the mit nn Philosophers menus in the Dinner menu is selected. The desired number of spaghetti philosophers is stored in the global variable iNumPhilosophers. The window is redrawn and the message WM_STARTDINNER is posted to the own window. The great meal begins once this message is received. The philosophers are first set to THINKING through the nState array and are then drawn with DrawPhilosoph(). Then the mutex for the state change is created and its handle stored in the global variable hMutexStateChange.

Only at this point are the individual philosophers created. A semaphore with a maximum counter of 1 and the current counter value of 0 is created. The returned handle is stored in the global array hSemPhilosophIsst. Then a thread is started for each philosopher with the start function fnPhilosoph(). The returned handle is stored in the global array hThreadPhilosophers. Each thread receives its own number as an argument with the fnPhilosoph() call, corresponding to its position within the various arrays (hThreadPhilosophers, nSpaghettis and nSemPhilosophIsst).

Finally, fnMenuWakeUp() is used to create another thread. The purpose of this thread is explained below. Let's first take a look at the start function of the professor threads, in fnPhilosoph(). It represents the identity of each philosopher, which begins its life in the THINKING state. A while loop is at the center of fnPhilosoph(). This loop is repeated until the professor has eaten all the spaghetti on their plate. The corresponding value in the nSpaghettis array has therefore reached 0.

The thread within this loop first uses the thread function Sleep() to pause itself. The sleep duration is determined by a random value. This means the thread becomes inactive for a certain amount of time. This time corresponds to the philosopher's THINKING state in which the professor is caught in lengthy esoteric conversation. It then calls TakeForks(), which tries to acquire the two forks that are needed to properly eat the spaghetti. When the function returns, the professor has managed to procure the second fork, so then Eating() is called. When this function also returns, the professor has eaten a portion of his or her spaghetti. The forks are put back down with PutDownForks(). This is done so the loop can be entered again (provided there is still some spaghetti) and another theory can be pondered (sleep).

The decisive factors at this point are the functions TakeForks(), Eating() and PutDownForks(). Since the functions are called by each professor thread, they receive the number of the professor that is making the function call. In other words, this is the corresponding index within the various global arrays as a function argument. This allows the function to distinguish between the different professors.

TakeForks() attempts to place its professor into the HUNGRY state through the status array. However, first WaitForSingleObject() forces it to wait for the handle in hMutexStateChange. Since all the functions that try to perform a write access on the status array do this, we're certain only one thread is modifying it at any time. This eliminates the possibility of a race condition. Once the thread has acquired the mutex, it places the professor into the HUNGRY state. The professor is then drawn in the new state with DrawPhilosoph(). It then calls Test() before releasing the mutex thereby permitting other threads to access the status array.

Test() is central in the back and forth of the forks. It's not only called from within TakeForks() but also from PutDownForks(). It checks whether the specified philosopher is hungry and whether their neighbors are eating at the time. If they aren't, then the specified professor can begin eating. At least two forks are available because neither neighbor is currently using their forks. The status of the hungry philosopher is therefore changed to EATING. At the same time, its semaphore from the global array hSemPhilosophIsst is again released using the API function ReleaseSemaphore(). This means that the semaphore is now in the signaled state.

The consequence of this action becomes clear after we return to the TakeForks() function. After the access mutex is returned by hMutexStateChange, the philosopher is redrawn by the function DrawPhilosoph(). If the Test() call was able to switch the professor to the EATING state, the appropriate character is displayed on the screen. If not, DrawPhilosoph() will show that the philosopher is still in the HUNGRY state.

The important step occurs at the end of the function. `WaitForSingleObject()` is called to specify the semaphore from the array `hSemPhilosophIsst` corresponding to the current philosopher. If the professor was able to be switched to the EATING state within `Test()`, its semaphore has been released and so is signalized. In that case, `WaitForSingleObject()` returns immediately to `TakeForks()`, so the latter can be ended and return to its caller. If not, the thread is blocked by the `Wait` call, so `TakeForks()` does not yet return.

The questions now are: How can `TakeForks()` be ended and how can the professor's semaphore be signalized so the `WaitForSingleObject()` call will return? The answer becomes clear when looking at the events after `TakeForks()` has successfully switched the professor to the EATING state. The function then returns to its caller, specifically `fnPhilosoph().Eating()` is called here. It first randomly determines how many pieces of spaghetti will be consumed. We then repeat a loop for each piece of spaghetti. The number of remaining pieces for that professor is decremented in this loop. The professor is redrawn and a short wait is introduced.

After the professor has eaten a certain number of spaghetti strands, `Eating()` returns the function `fnPhilosoph()` to its caller. `PutDownForks()` is called from there since the professor is about to enter another state of deep thought before eating more spaghetti.

`PutDownForks()` first acquires the mutex from `hMutexStateChange`. It can then immediately switch the current philosopher to the THINKING state and redraw him or her with a `DrawPhilosoph()` call. The decisive step occurs next. This step also answers the question posed above about how `TakeForks()` can be ended if its `Test()` call was not able to acquire two forks for its professor. `PutDownForks()` now calls `Test()` twice. It calls it once for the left neighbor and once for the right. If the left or rather the right neighbor is hungry and if their neighbors (neighbors to the left and right) are not in the EATING state (which will definitely be the case for one of the neighbors — the professor from `PutDownForks()`), then that neighbor is switched to EATING and its semaphore released (signalized).

A professor thread that was blocked in `TakeForks()` because both of its neighbors were EATING can be released if one of its neighbors calls `Test()` within the `PutDownForks()` function. This is the decisive 'trick' of the application. Because a blocked professor thread is unable to acquire forks, the poor professor would eventually starve. Since it's other professor threads that release the waiting professors, this application requires using semaphores rather than mutexes. After all, mutexes can only be released by their owners, whereas semaphores can be released by other threads as well.

So in this way, all the philosophers will eventually have some spaghetti. Once a philosopher has eaten all the spaghetti on their plate, the `fnPhilosoph()` function ends. The thread is thereby terminated. One after the other, all the philosopher threads thus meet are terminated.

One more thread comes into play. This thread was produced at the start of the simulation and is executed in the function `fnMenuWakeUp()`. This function first makes the Dinner menu inaccessible. This prevents the user from selecting another menu during the simulation. Then it uses `WaitForMultipleObjects()` to wait until all the philosopher threads have been ended. It does this by simply providing the number of lunchtime philosophers to `WaitForMultipleObjects()` and a pointer containing the handles to the philosopher threads to the `hThreadPhilosophers` array.

Once `WaitForMultipleObjects()` returns, all the dinner plates are empty. `fnMenuWakeUp()` deletes all the thread handles and releases the Dinner menu again.

You'll find the following program(s) on the companion CD-ROM



DINNER.C (C listing)
DINNER.RC (C listing)
RESOURCE.H (C listing)



Virtual Memory Management

If Windows 95 has truly earned the title “32-bit system,” it’s probably because of its virtual memory management. Compared to previous versions of Windows, the virtual memory manager in Windows 95 was completely rewritten. It now finally uses the full potential of the modern Intel microprocessors.

Windows 95 no longer includes many components that frustrated users. Real mode now only appears in DOS windows. The separate segment and offset addresses, combining FAR pointers, and hassling with memory segments and segment registers is history for Win32 applications. Previous Windows applications were able to access the entire range of memory and thereby inadvertently crash the system. This is now a thing of the past. In Windows 95 an application only sees a limited section of the available memory (i.e., its address space). Its universe ends at the edges of this 4 Gigabyte space. It cannot access additional memory either by accessing data through pointers or by using CALL or INT processor commands.

It’s obvious that this makes the system considerably more stable and reliable. An application can still crash but no longer can it cause other applications to crash. Even the operating system kernel is protected against attack. Although this apparently tightly woven net still has some holes, you have to use a crowbar to open these holes because normal program code cannot escape from its address space. Things like accidentally picking up a null pointer, skipping around some 10 Meg of reserved address buffer: none of these kinds of problems will cause a Windows session to crash any longer.

Virtual Memory Management Basics

The key terms associated with virtual memory management are “flat memory model” and “protected mode.” Each application (i.e., each process) is given an address space among the potential 4 Gigabytes in size that is used by both the processor and the operating system. 4 Gigabytes corresponds to the number of memory locations that can be addressed using the 32-bit address registers in the Intel 80386 chip and its successors. Separate offset and segment addresses are not required anymore for accessing the memory. All that’s needed is a single 32-bit address. This is, technically, an offset address in the application’s address space. However, since segments no longer exist, we no longer say “offset.” Instead, we simply talk about the “address.” This is, of course, an offset.

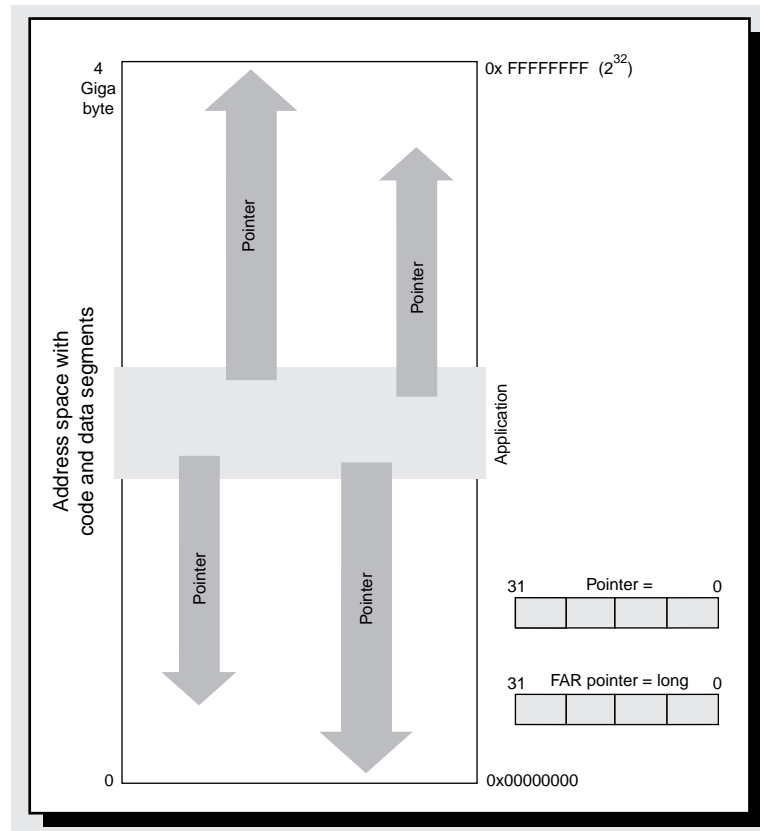
C Pointers in Win32

32-bit pointers are stored in long variables at the C level. No difference exists among FAR pointers, NEAR pointers, BASED pointers, etc. Only one type of pointer is available now. It represents an offset in the application’s address space. Therefore, you can copy macros such as MK_FP for joining FAR pointers. This avoids the hassle of segment and offset addresses when running through a large array (larger than 64K). Once you have a pointer to an array, use it to step through the memory. It’s irrelevant if the array is only a few bytes long or if it’s several hundred megabytes long.

By convention, FAR pointer variables start with the characters lp according to Hungarian notation. Now they just start with the character p. So, lpBuffer is now pBuffer. However, you’ll often see lp in include files and help systems until somebody changes them. Don’t let that bother you; just forge ahead knowing that this is just a standard 32-bit pointer.

When an application is started, the system kernel creates an address space 4 Gigabytes in size. It also sets the base register for program code and data at the beginning of this memory range. Therefore, a pointer doesn’t have to contain segment components any more. Now there is either only a data segment or code segment. A 32-bit offset address is enough. Also, the context always indicates whether the pointer is pointing to the code segment or the data segment.

Accessing the address space of an application using pointers



Compilers do more than just make it easy to convert high level language programs into machine language. They also make the program code more efficient. This is just one of the reasons true 32-bit applications are normally faster than 16-bit applications.

Separation of system code and user code

In addition, you can no longer modify an application's segment register because it's controlled by the operating system. This is accomplished using a second feature of the protected mode: separation of system code from user code. All Intel processors starting with the 80386 let you organize program modules in one of four priority levels called *rings*. Ring 0 is reserved for the operating system kernel. Rings 1-2 are reserved for system components that serve as go-betweens between the system kernel and the application programs. Ring 3 is reserved for applications.

Accessing certain processor registers becomes increasingly more difficult the further you are from Ring 0. This is also true when executing some machine language instructions. The idea behind this is that only the system kernel is allowed to affect the system elements that are required for the system to run safely. For example, only program code in Ring 0 can access processor registers that are responsible for how the application's address space is organized. The same is true for accessing the hardware using the I/O ports. An application can be denied access to the hardware in Ring 0 if the kernel considers this is appropriate.

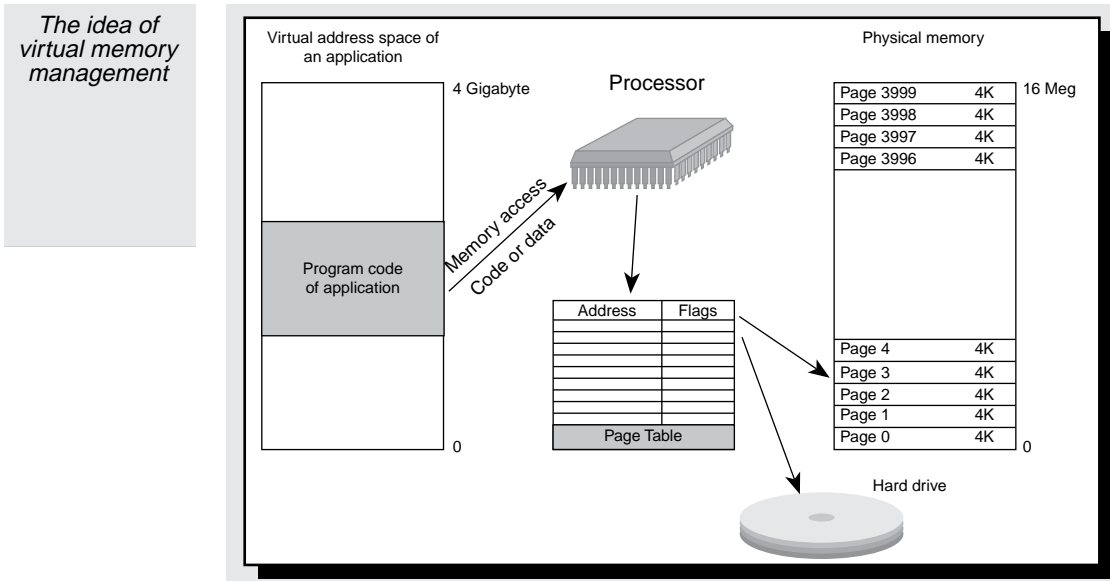
The processor uses the four rings to create a contrast between the different elements of an operating system and its applications. However, only the two outermost rings are used in Windows 95. Most of the operating system runs in Ring 0. Although a few system components run in Ring 3, this is also where the applications run. This way the operating system controls their execution. The operating system can, for example, block them from changing their address space so they can't conflict with other applications and the core itself. So, the processor makes certain the system code in Ring 0 always has greater leverage.

Virtual Memory Management

The significance of this is that an application never recognizes that its 4 Gigabyte address space isn't even filled with physical storage space. Of course, how could it, since most PCs don't have more than 8 or 16 Meg on board? Instead, the operating system only fills areas of the address space with memory where the program code and the associated data are loaded. If an application requires additional memory when it's executed using appropriate system services, the memory block provided by the system is filled with physical storage memory.

Only by using virtual memory management can a program use more memory than is available in this manner. Although it's controlled by the system core of Windows 95, it's based on a very basic mechanism of addressing memory. This mechanism must already be implemented at the processor level. It also must be available in Intel chips since the 80386. Several questions arise if you consider that only now are the leading desktop operating system using these capabilities. This is fully ten years after the 386 generation of chips were developed.

The idea behind virtual memory management is that an application's entire virtual memory doesn't have to be physically present. Tests have shown that an application doesn't permanently access its entire memory. Instead, it limits itself primarily to the regions containing its code and data. Code profilers show this very nicely. So, we use the hard drive to swap out physical memory and to load it back in again when necessary. (In this case, "necessary" means when the application accesses it.) This memory always appears to be present according to the application. The application isn't even aware that the processor is intercepting the request to access a range of memory that has been swapped out to the hard drive and has to first reload it back into physical memory. In this way we can allocate much more memory to two applications running concurrently than fits in the computer's RAM. The hard drive becomes a natural extension of the RAM.



For efficient loading and unloading of memory blocks, the processor divides the virtual memory in 4K segments called *pages*. Whenever the machine language of an application attempts to access the application's address space in any manner, the processor first uses the indicated memory address to calculate the page to which this memory block belongs. It uses the page tables or page directories to do this. An application's address space is distributed piece by piece across pages in physical addresses using these page tables. Although the processor prescribes the format of these tables, it is the operating system that manages them.

So the processor looks in the appropriate entry in the page table to see whether the page in question is in physical memory. An entry in a page table has special flags for this that provide the necessary information. If the answer is yes, the processor can access the memory block from the appropriate page. If the answer is no, the processor reports this to the operating system core. The core promptly loads the correct page from the swap file on the hard drive.

Then it's available to the application. If all the available physical memory is being used, at least one page is unloaded from memory and written to the swap file. It's preferable that several pages are unloaded at the same time. The maximum size of the swap file is all that limits the amount of virtual memory available for all active processes. Here, you can see how the hardware and software work hand in hand. At one level, the processor checks the memory address being accessed. It is not, however, concerned with the actual swapping of the memory. This task belongs to the operating system.

There is an interesting side effect of dynamically loading and writing pages. What the program views as a contiguous address space is instead spread over many pages. These pages reside at very different addresses in the physical memory. The memory for an application might be fragmented almost to the point of not being recognized. However, that is all invisible to the application. It can never access memory without the processor detecting it and directing the access to the appropriate page. So, what the application sees is an artificial, or virtual, construct that never takes place without the processor participating.

All of today's microprocessors support the idea of virtual memory management using pages. This is the only way that Microsoft is able to port Windows NT to different platforms. All that is different among the various CPUs is the way they use the registers and operations to manage page tables and the size of the pages themselves. With Alpha processors, the page sizes are 8K in size. The page sizes with MIPS, PowerPC and Intel are 4K.

Naturally, swapping memory in and out is sometimes a time consuming process. Also, accessing the hard drive requires more time compared to accessing memory. So, it's no small task for the system to always have memory available for an application. For example, pages from other applications that haven't been used for a long time are automatically swapped out. Other pages are then swapped back in when the system recognizes that they are being accessed. The multitasking system in Windows 95 allows this process to run in the background without affecting the work of the user.

Makeup of the address space

If a process is always limited to its address space when accessing memory, how can it access the functions of the operating system that lie outside this address space? Also, how can it call either its functions in DLL files or the system's functions in DLL files? The answer is found by looking at the "road map" of the address space of a 32-bit application.

New math

It takes a while to become familiar with the big numbers. In the early days of real mode, no one ever heard of addresses with eight hex digits. You knew that 0xFFFF was the highest value of the 32-bit world. Also, you hardly needed to know any hex numbers at all. However, when you see something like 0xCFFFFFFF you have pause for a short time to determine where it's located.

0x00000000 - 0x00000FFF

The first 4K of the address space of a process are reserved. They are used to capture pointer operations with NULL pointers, which is a common error in C programs. When an application accesses the memory using a NULL pointer, the application goes into this range of memory, and calls the operating system in the process. It responds by invoking an exception (you will read more about exceptions in the "Exceptions" section).

Of course it would be enough to protect the lowest byte, or DWORD. However, because the protection must be done using the processor's page tables, you have to block out at least one page (4K with the Intel processors).

```

void test( long l )
{
    long *pl;
    pl = malloc( sizeof(long) );
    // if Malloc returns NULL, there's a bang/crash here!
    *pl = l;
}

```

0x00000000 - 0x003FFFFFFF

This range extends to the 4 Meg point. It's reserved for executing DOS and Win16 applications. A Win32 program shouldn't access this area.

0x00400000 - 0x7FFFFFFF

The area between 4 Meg and 2 Gigabyte is where the actual application or the process it started is located. This is also where all the memory is stored that the application or one of its DLLs is dynamically allocated during execution.

0x80000000 - BFFFFFFF

The segment from 2 Gigabyte to 3 Gigabyte is reserved for those DLLs and their data which are used jointly by the different applications in memory. After the DLLs are inserted into an application's address space, the program code can invoke the desired DLL functions. An application can read the contents of this section. It should, however, avoid any writing access, if possible.

0xC0000000 - FFFFFFFF

The highest Gigabyte in the address space is also used for inserted program code and data that are used jointly by all the processes which the system is running. This is where the system device drivers are located, as well as components of the memory management machine, and program code for the file system.

One page in several address spaces

The idea of page tables provides interesting side effects that are not at first obvious. For example, the system core can use the page tables to insert a page in several address spaces. The result of this is that several processes access the same page without even knowing it. They even see the page at completely different positions in their address spaces. This is used especially for program code. The memory it uses is not written to but only executed or read. If we used this procedure with pages containing data that are to be written, such as pages from the pool for the heap, the programs would overwrite each other's memory area.

Organization of the address space of a process		
Address (Hex)	Range	Contents
FFFFFFFFH	4 Gigabytes-1	Jointly used system area Ring 0 - Software System DLL's Device drivers
C00000000H	3 Gigabytes	Region used by all applications 16-bit applications Application DLL's Objects
BFFFFFFFHH	2 Gigabytes	Private region of a process
800000000H	4 Megabytes	32-bit Windows applications
7FFFFFFFHH	1 Megabyte	Usually not used
004000000H		
003FFFFFFFHH		
0010000000H		
000FFFFFFFHH		
0000000000H	0	MS-DOS region

Exceptions

Exceptions are triggered by the processor whenever a block of program code is executed which attempts to execute an illegal operation. These include accessing memory that is not enabled or dividing by zero. The operating system can intercept this exception and terminate the application that is causing it. However, this is not desirable and is why applications have their own exception handlers. These handles allow an application to remove the cause of the error (for example, by making more memory available) and then to continue executing the program as usual. It's similar to an On-Error-Goto in C. However, this isn't a problem with program code because this code is not written to but only read. So, the technique we just described is used wherever the same block of program code is used concurrently by several processes. This is especially true for the following two situations:

Multiple sessions of an application

When an application is started more than once, the same code has to be executed by multiple processes. So why load the code more than once? A single copy for all sessions should be enough. However, the global data for the individual sessions and the dynamically allocated memory must be managed separately for all sessions of an application. Otherwise, they would lose their individual character.

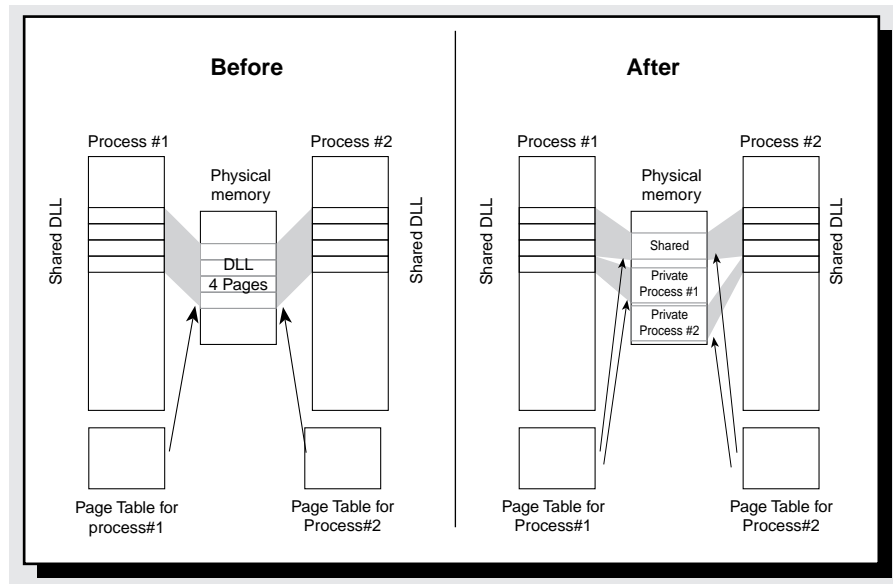
DLLs and System Code

If several processes are running in parallel, some DLLs are usually being used simultaneously by several processes. These processes include the system DLLs, KERNEL32 and usually a few others. Their code usually needs to be loaded only once. However, this does not count for the pages with writeable data, which used to be called a data segment. First, they give each DLL session its individual character, which develops from the communication between the process and the DLL. Keep in mind that processes call the DLL functions in a different sequence and with different parameter values. Therefore, the data of the separate DLL sessions is likewise different. That has to be maintained.

WRITECOPY only with NT

When Windows 95 recognizes that several sources can write to a page, the system core creates a separate copy of this page for each DLL copy. An example of how Windows 95 can recognize this is perhaps the page belongs to the global variables of a DLL used by multiple applications. This doesn't happen immediately; only when a new process starts that use this DLL. That is very farsighted, sometimes even too farsighted. Naturally, situations may develop where a long time goes by before the first process writes to this page. Only then does the process need to have a private copy of the page.

*Write On Copy
saves memory
space*



The process supports this write on copy process. Whenever a process writes to a page with the attribute `PAGE_WRITECOPY`, a private copy of this page is established for the process. The page is then removed from joint use by other processes. So, each process that changes a memory segment is given its copy of the affected pages. It does this while it continues to share other pages with other processes.

Although Windows NT uses this feature extensively, Windows 95 does not. If a page could be overwritten differently by several copies of a piece of code, when the code is generated, separate copies of the particular pages are immediately created. These copies are initialized with the page contents. At this point, the individual copies of this page go their separate ways. We don't know whether the Windows 95 had problems working with `PAGE_WRITECOPY`. Perhaps it was simply a victim of the forced delimitation regarding Windows NT. It is true, however, that Windows 95 also does not support some other page attributes which Windows NT understands. In actual practice, though, there are no disadvantages presented to the developer as a result.

Page attributes

Pages have different attributes that depend on how they are used. Pages can be read and written to if they're used for data. Others contain code so they should only be executed. Therefore, all processors that support virtual memory management provide a way of allocating attributes to a page. It's therefore obvious who has access to the page and what type of access they have. Windows NT and Win32 API functions add several of their attributes to those provided by the processor. The number is increased to a total of ten attributes. Only three of those are available in Windows 95: `PAGE_NOACCESS`, `PAGE_READONLY`, `PAGE_READWRITE`, but they are perfectly enough.

PAGE_NOACCESS

Every access is handled as an exception. The processor acts immediately, whether an attempt is made to read, write or execute. This attribute is used for pages that are not located in memory.

PAGE_READONLY

The page can only be read. Any other access generates an exception.

PAGE_READWRITE

It's possible to read and write to the page.

PAGE_WRITECOPY (NT only)

Read and write access is allowed. The process is given a private copy of the page when write access occurs.

PAGE_EXECUTE (NT only)

The page can be executed. However, it's not possible to read or write to the page. That generates an exception.

PAGE_EXECUTE_READ (NT only)

However, it's not possible to read or write to the page. This is useful whenever a block of code needs to reproduce itself or another block of code. To do this, it must read the code before it can write it to another page.

PAGE_EXECUTE_READWRITE (NT only)

The complete menu: The page can be read, written to, and executed.

PAGE_EXECUTE_WRITECOPY (NT only)

The page can be read, written to, and executed. However, when write access occurs, a private copy is produced for the particular process.

PAGE_GUARD (NT only)

Guard pages are used to monitor objects such as stacks and heaps that dynamically expand the memory they use. They act somewhat like a protected perimeter that activated a signal as an exception when it is touched. For example, the stack manager uses this to make more memory available for the stack when the stack goes to the next page using a stack operation (PUSH).

After the exception is generated, the guard attribute is automatically removed. No more exceptions can then be generated by an additional access.

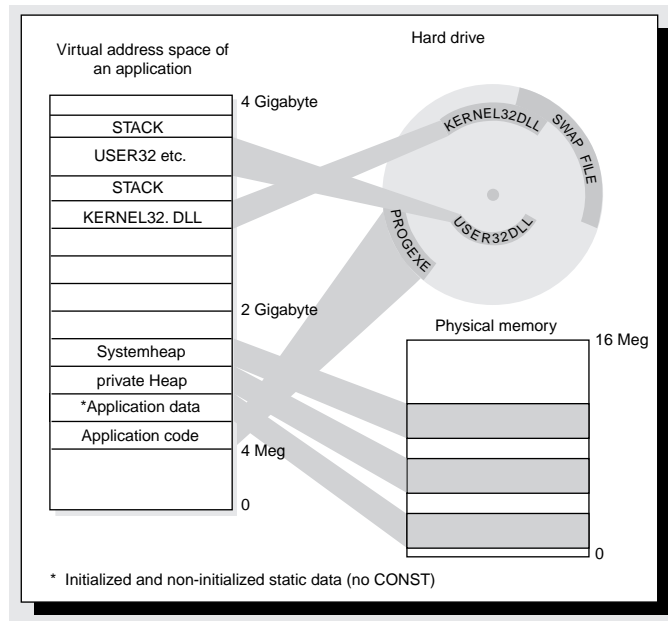
PAGE_EXECUTE_NOCACHE (NT only)

Designed especially for device drivers that access memory that is mapped to the processor's RAM as the result of a hardware expansion, such as a graphics card. These pages are not swapped in or out, but are taken "as is."

Memory Mapped Files

An additional idea of virtual memory management is created after the program code and DLLs are inserted in an application's address space. This is the memory mapped files. Since we cannot write to the pages of a DLL anyway, they always simply correspond to the content of the respective DLL file on the hard drive. So, why store these pages in the swap file if you can simply reload them directly from the appropriate file if needed? This saves the time consuming process of storing a copy in the swap file. You also save room in the swap file.

The principle of memory mapped files



This is where the idea of memory mapped files occurs. Win32 API functions are used to map a file to an area of the virtual memory. After that, you simply act as if the entire file was already in memory, which is not correct. For one thing, no page at all from the file is in memory. Pages are not loaded from the file until a concrete accession occurs and if they are not already located in memory. This way, the program loader doesn't even have to load the complete program when it is started; it's enough to just map it to memory. Once the execution has started and the processor runs through the program code, it generates an exception when it comes to a page that hasn't been loaded yet. So, the appropriate page (and usually a couple more at the same time) is loaded.

Therefore, any routines that are never used do not have to be loaded. One result is a noticeably increased speed when starting an application. Even application programs can use memory mapped files to simplify file processing. Also, for more information about the role of the processor in virtual memory management, see Chapter 35. It's interesting to see how the processor and the operating system work together in this case.

Accessing Memory Using Win32 API

Developers have five options available in Win32 for dynamically allocating memory while their application is executing. Each method is associated with a set of API functions. Functions with approximately the same names and having very similar purposes are used.

Several reasons explain why so much variety is needed. Two interfaces, for global and local memory, correspond to the known 16-bit interfaces with functions like `GlobalAlloc()` and `LocalAlloc()`, `GlobalLock()` and `LocalLock()`, etc. These functions use ideas that existed before Windows 95. However, they're implemented in Win32 API so it's easier to port existing 16-bit applications. Win32 API doesn't distinguish between local and global memory any more; it satisfies the memory requirements of these functions from normal virtual memory. Therefore, the local and global functions were mapped internally to the new virtual functions of Win32 API. However, you can continue to call them as you have done in the past. The different internal memory management can't be seen externally. This means that existing applications should be readily compiled and executed in Win32 without redesigning the program's memory management at the same time. Indeed, the new memory management is much more flexible. An application is no longer required to lock memory before accessing it (`LocalLock()` or `GlobalLock()`). Therefore, the new functions no longer require a handle to identify a block of memory when you don't even know just where it's located in memory. The pointer is all you need. Some it might be worth it to adapt old 16-bit calls to the new conditions of virtual memory management to acquire more space.

Global memory and local memory are two interfaces of virtual memory management. The third virtual memory management interface is the Virtual... functions like `VirtualAlloc()` and `VirtualQuery()`. They represent the preferred interface for an application to use for allocating memory under Win32 API. However, the Virtual... functions are only designed for relatively large blocks of memory. Examples are those needed for processing complete files or extensive arrays in memory. Virtual memory is always allocated on the level of pages so no block can be smaller than 4K. For that reason, this interface usually isn't well suited for the dynamic management of several small objects such as when you're creating tree structures and lists. In that case, you would use the different Heap... functions, which allocate memory a byte at a time. They represent the fourth interface that memory management provides.

The fifth interface is also designed more for small objects. However, instead of originating from Win32 API, it comes from the runtime library of the C compiler. This includes specifically `malloc()` and the associated functions `realloc()` and `free()`. The problems that `malloc()` had in Win16 are not present in Win32. This is a simple method for dynamically allocating memory to several megabytes, to change the size of the memory and to free it if it's no longer required.

Information on the following pages describe how to work with virtual memory and how to manage heaps. First, the following tables summarize the Win32 API functions for managing memory.

Win32 functions for memory management	
Win16 functions that are still supported for global memory	
Function	Task
<code>GlobalAlloc</code>	Allocates global memory and returns a handle for it.
<code>GlobalDiscard</code>	Discards the contents of a memory block.
<code>GlobalFlags</code>	Provides information about a previously allocated block from global memory.
<code>GlobalFree</code>	Frees up a block which was allocated with <code>GlobalAlloc()</code> .
<code>GlobalHandle</code>	Returns the handle for a previously allocated block from global memory.
<code>GlobalLock</code>	Locks a previously allocated block in memory.
<code>GlobalMemoryStatus</code>	Provides information about the available memory.
<code>GlobalReAlloc</code>	Enlarges or reduces a previously allocated block.
<code>GlobalSize</code>	Returns the size of a block which has been allocated.
<code>GlobalUnlock</code>	Unlocks a block so that it can be moved again.

Win32 functions for memory management (continued)	
Working with blocks of memory	
Function	Task
CopyMemory	Copies a memory block.
FillMemory	Fills a memory block with a constant byte.
MoveMemory	Moves a memory block.
ZeroMemory	Fills a memory block with zeroes.
Win-16 functions for local memory that are still supported	
Function	Task
LocalAlloc	Allocates a block from local memory.
LocalHandle	Returns a handle for a block from local memory.
LocalLock	Locks a block previously allocated from local memory with LocalAlloc() so it can't be moved.
LocalReAlloc	Enlarges or reduces a block previously allocated in local memory.
LocalSize	Returns the size of a block previously allocated in local memory.
LocalUnlock	Unlocks a block from local memory so that it can be moved again.
Functions for managing heaps	
Function	Task
GetProcessHeap	Returns a handle for the heap of the current process.
GetProcessHeaps	Fills an array with the handles of all heaps of the current process.
HeapAlloc	Allocates memory from the heap.
HeapCreate	Creates a new heap.
HeapDestroy	Deletes a heap which has been created.
HeapFree	Frees a block of memory which was previously allocated by a heap.
HeapReAlloc	Enlarges or reduces a block which was previously allocated by a heap.
HeapSize	Returns the size of a previously allocated heap.
Functions for managing virtual memory	
Function	Task
VirtualAlloc	Allocates a block in virtual memory or makes it accessible.
VirtualFree	Frees a previously allocated block from virtual memory.
VirtualProtect	Protects the page attributes for pages in virtual memory.
VirtualProtectEx	Like VirtualProtect(), but it also makes it possible to protect pages in other processes.
VirtualQuery	Provides information about the attributes of the pages in a memory block.
VirtualQueryEx	Like VirtualQuery(), but it enables you to query pages in other processes.

Virtual Memory Management

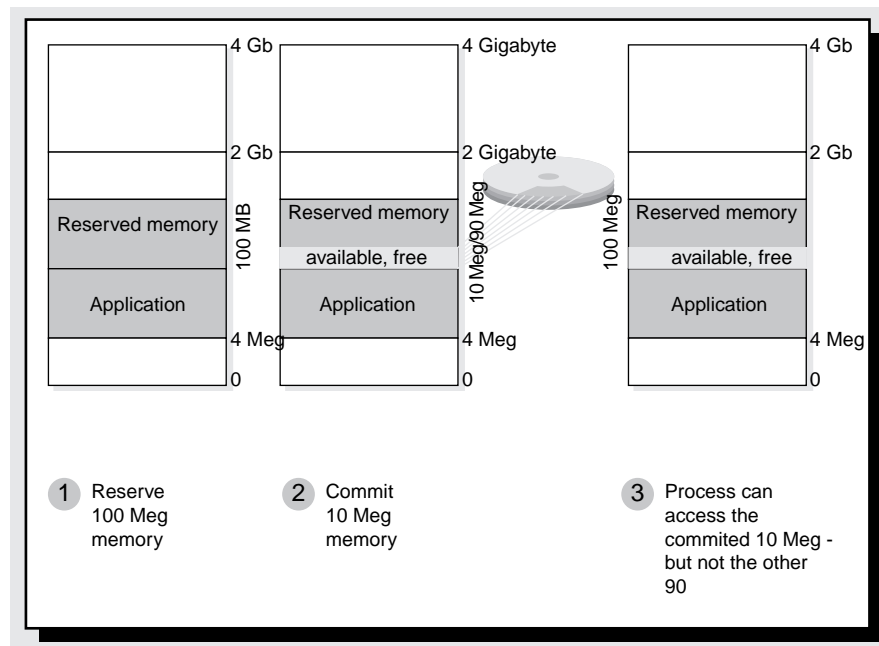
A complete set of functions whose names start with Virtual provides access to virtual memory management for an application. More is involved than simply allocating a few bytes and adding a structure. Sectors in the address space must also be reserved so they aren't used up by other processes. It doesn't matter at this point how and when they will be used later. For now, we want to make certain the sectors are reserved from the start.

The system uses this opportunity to reserve parts of the address space for itself and for the loaded process. However, this also benefit the user code; namely anywhere where data and data structures dynamically expand during execution of the program. This usually requires a great deal of work without a virtual memory manager. The add-on memory can't simply be added at the back end when a structure expands. This unnecessarily complicates everything. You either must increase the use of pointers and linked lists or you must let the memory segment move when the structure expands.

It's simpler with virtual memory management. First, estimate how much space you'll need for the different dynamic data structures in your program, preferably separately for each one, since you'll be allocating space separately. How big are the individual data structures? What is the maximum number? Fortunately, you can be generous with your calculations. You know that you will never add 123,000,000 records to your video database. However, the virtual memory manager doesn't know that. Although it doesn't make sense, you could create regions several hundred megabytes in size without any problems. Still, enough virtual memory is available. It's taken from the memory left over between 4 Meg and 2 Gigabyte that the process isn't using. So, besides a program a few K in size, enough room is available for several hundred megabytes in arrays.

Yet allocating the memory and using it are different things at this level. Often the allocated regions are much too large for the operating system to provide physical memory for them. Besides, that would have an important consequence. Corresponding memory would must be reserved in the swap file. After all, the pages of the array must be written out whenever their memory space is needed by something else. Often a reserved range doesn't even fit completely in the physical memory so nothing works without swapping. In some cases the new pages in the swap file must be initialized. At the latest, this situation would expose this kind of interaction with virtual memory as a farce. You would have the slowest 32-bit operating system the PC world has ever seen.

Allocation of the process memory on the basis of the Virtual functions



So it won't work that way. Memory has to be reserved so it can be used, and it also has to be committed. That is, a process makes clear that it wants to use this memory now. This is when the virtual memory manager makes room available in the swap file for the pages inside this range. At the least the virtual memory manager makes room for those pages for which this hasn't been done yet. Whenever you want to expand an array or another dynamic data structure during execution of a program, tell the memory manager how much memory you need from this point. Problems can occur if you miscalculate and in the process of writing out the data, you end up in a block of memory that hasn't been allocated yet. The processor will register an exception regardless of whether this block is within an array, the memory of another array or in the empty space between the reserved memory blocks. The program might be terminated if there is not exception handler.

Virtual functions

VirtualAlloc() is the starting point for working with virtual memory. It's used to reserve memory blocks within the virtual address space. VirtualFree() is used to free them later. VirtualAlloc() is also used to commit those parts of a reserved block that you want to use. VirtualProtect() and VirtualQuery() are also used to query the attributes of pages very specifically and to protect them. This is especially useful for internal applications.

Win32 functions for memory management	
Function	Task
VirtualAlloc	Allocates a block in virtual memory or makes it accessible.
VirtualFree	Frees up a previously allocated block from virtual memory.
VirtualProtect	Protects the page attributes for pages in virtual memory.
VirtualProtectEx	Like VirtualProtect(), but it makes it possible to protect pages in other processes.
VirtualQuery	Provides information about the attributes of pages in a block of memory.
VirtualQueryEx	Like VirtualQuery(), but makes it possible to query pages in other processes.

With the different virtual function, the pages in a process's address space go through three conditions:

Free

The page is neither reserved nor committed. Access to it is not allowed. It can be reserved for an application together with other pages if the program required memory.

Reserved

The page was reserved as part of a block for a call and therefore it can't be given to any other calls. Access to the page is not allowed.

Committed

The page was reserved as part of a block and was physically committed by a page in the swap file. Access is always possible.

VirtualAlloc()

All pages start out free in the original state of a process. More pages are reserved when the process begins. Some of pages are committed for the system code that is to be inserted, the required DLLs and finally for the application itself. Once it is left to itself, the process can use VirtualAlloc() to acquire memory as it executes.

```

LPVOID VirtualAlloc(
    LPVOID lpAddress,           // Starting address of desired area
    DWORD dwSize,               // Size of area
    DWORD flAllocationType,     // Order to VirtualAlloc()
    DWORD flProtect             // Page attributes
);

```

VirtualAlloc() shows something that is true for most of the other virtual functions: The address of the memory block to be dealt with is the first parameter that is expected. VirtualAlloc() uses this parameter to supply the base address of the desired block. This option is mostly designed for the system when it is necessary to reserve certain blocks of memory whose location is prescribed, for example the region above 4 Meg. Although user code can also indicate a desired address here, it's uncertain whether it is committed. If it points to a block that is already reserved, the reservation fails. So, the simplest thing is to use NULL for this parameter and let the function make the selections.

The second parameter that is expected is the size of the desired block. The third parameter is a flag appearing as a constant. It represents a type of instruction for VirtualAlloc():

MEM_RESERVE

Only reserve the memory; don't commit it yet.

MEM_COMMIT

Commit the memory, i.e., make appropriate room available in the swap file. You can combine both flags so the desired block is not just reserved, but also committed at the same time. However, this can require some time, depending on the size of the block. The hard drive starts working feverishly during this time.

The last parameter is the page attributes you want for the block. Select one of the three following settings in Windows 95:

PAGE_NOACCESS

Even if the page is committed, no one can access it. This is useful if you want to keep a block available but you want to block any access to it for now. If you want to make it accessible later, use `VirtualProtect()` to give the block different page attributes and free it.

PAGE_READONLY

Use this attribute for memory that is read-only. Program code is executed when it's accepted by the memory.

PAGE_READWRITE

The pages can be read or written to at any time.

If the desired memory was allocated, `VirtualAlloc()` returns the starting address of the block as a `VOID` pointer or with `NULL` in case of an error. Errors can occur if the requested block of memory is exceedingly large. Errors can also occur if a process allocated so much virtual memory the process memory in the address space can't provide any room for additional memory. It's also critical to indicate a predefined starting address when you call `VirtualAlloc()`. In this situation, expect the function returning `NULL` because at least part of the indicated block was previously reserved.

Look at the address that is returned. Notice that `VirtualAlloc()` always starts the block of memory that is returned at a 64K boundary. That has to do with the internal management of blocks in the system kernel. The address is simplified to multiples of 64K because the lower 16 bits of a block's starting address are always 0 then. You only need to store the upper 16 address bits. One word instead of a `DWORD`. Sometimes that can make a big difference. Also, the size of the reserved block is always rounded off to the size of a page since the reserved memory is reserved as pages. Even if you only use one byte in the last page, it is reserved as well. It can, however, be used to the fullest when it is committed.

Once you have successfully reserved a block of memory, you can commit parts of this memory later by calling `VirtualAlloc()` again. You don't necessarily need to use the pointer to the starting address of the block, which is returned by `VirtualAlloc()`. Any particular subrange within a reserved range can be committed as well. However, that is usually only helpful if you aren't using the block of memory in a continuous fashion. Then new data are not necessarily simply attached to the previous data.

However, as a rule you will expand the block of memory in a continuous fashion. To free new memory at the end of what has already been committed, use a `VirtualAlloc()` call to indicate the base address of the memory. Use the total length of the continually used portion from the beginning of the block of memory for the length. This affects all pages in which even a single byte of the indicated block is found. Although the pages at the beginning are included in the call repeatedly, this won't matter to the function. This has no effect on blocks that are already committed. It does involve a matter of performance, however, as we will show in a later section by using a sample program.

VirtualFree()

Virtual memory that was reserved or committed using `VirtualAlloc()` can be freed using `VirtualFree()`, either the complete block or a subregion of the block.

```

BOOL VirtualFree(
    LPVOID lpAddress,           // Starting address of area to be released
    DWORD dwSize,              // Size of area
    DWORD dwFreeType            // Order to VirtualFree()
);

```

To free the memory, use the starting address that you were given when you called `VirtualAlloc()` as the first argument. The second parameter must be a 0. The third argument is the constant `MEM_RELEASE`. Notice that only complete blocks of memory are freed this way. It's not possible to free individual subregions or a segment at the end.

Freeing up a block of memory fails if all the pages within the block are not in the same condition, i.e., either committed or not committed. If part of a block is committed, you'll first need to call `VirtualFree()` to make it unusable. In this case, instead of `MEM_RELEASE`, use `MEM_DECOMMIT` and the starting address of the block and its size in bytes. `MEM_DECOMMIT` makes certain that no space in the swap file is kept committed for the pages. The results of the function will tell you whether the attempt was successful. In this case, the function returns `TRUE`. Otherwise, use `GetLastError()` for more error information.

If you can't determine which pages were committed and which ones weren't, execute a `VirtualFree()` call with `MEM_DECOMMIT` and indicate the entire block. If you then issue a call with `MEM_RELEASE`, nothing else can go wrong.

Changing and querying page attributes

Functions `VirtualProtect()`, `VirtualProtectEx()`, `VirtualQuery()` and `VirtualQueryEx()` can be used to explicitly protect or query the attributes of pages. Although normal applications don't have to refer to them, they're occasionally important for internal applications.

While the `VirtualProtect()` functions are used to protect page attributes, the attributes can be queried using the `VirtualQuery()` functions. Either case also has an additional "Ex" function that can be used to access memory for another process besides the current process memory. You should use great caution when using these functions since they can easily crash another process. The process handle is used to identify the process that is being addressed.

```
DWORD VirtualQuery(
    LPCVOID lpAddress,                // Starting address of area
    PMEMORY_BASIC_INFORMATION lpBuffer, // Pointer to Detailinfos
    DWORD dwLength                    // Length of structure with Detailinfos
);

DWORD VirtualQueryEx(
    HANDLE hProcess,                 // Process whose pages are to be queried
    LPCVOID lpAddress,                // Starting address of area
    PMEMORY_BASIC_INFORMATION lpBuffer, // Pointer to Detailinfos
    DWORD dwLength                    // Length of structure with Detailinfos
);
```

`VirtualQuery()` and `VirtualQueryEx()` expect the starting address of the block whose page attributes are to be read. The desired information about the pages is stored in a data structure of data type `MEMORY_BASIC_INFORMATION`. When the function is called, the calling program must indicate a pointer to this kind of structure as the second parameter. The size of the structure being transferred is expected in the third parameter. This parameter must be initialized with `sizeof(MEMORY_BASIC_INFORMATION)`. This lets the function verify whether enough memory is available for the information to be stored.

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;                // Starting address of area
    PVOID AllocationBase;             // Starting address of page where the area begins
    DWORD AllocationProtect;          // Page attributes
    DWORD RegionSize;                 // Size of area with similar page attributes
    DWORD State;                      // Status (Free, Reserved, Available)
    DWORD Protect;                    // Page attributes again
    DWORD Type;                       // Origin, Win95: always MEM_PRIVATE
} MEMORY_BASIC_INFORMATION;
```

In this structure, the function provides information about the attributes of the pages in the indicated region. It does so for all pages having the same page attributes beginning with the starting address. By saying "the same" we mean they all are free,

reserved, or committed. They also have page attributes in common, i.e., either `PAGE_NOACCESS`, `PAGE_READWRITE` or `PAGE_READONLY`. Whenever the `VirtualQuery()` function finds a page with different attributes, the search stops and the function places the information about the block of similar pages that it found in the `MEMORY_BASIC_INFORMATION` structure above it. The result of the function is the length of the bytes loaded with information in the `MEMORY_BASIC_INFORMATION` structure.

The first two fields of the region in question contain the base address (`BaseAddress`) and the starting address of the page (`AllocationBase`) in which the `BaseAddress` is located. If the `BaseAddress` points to the beginning of a page, its contents are identical to those of `AllocationBase`.

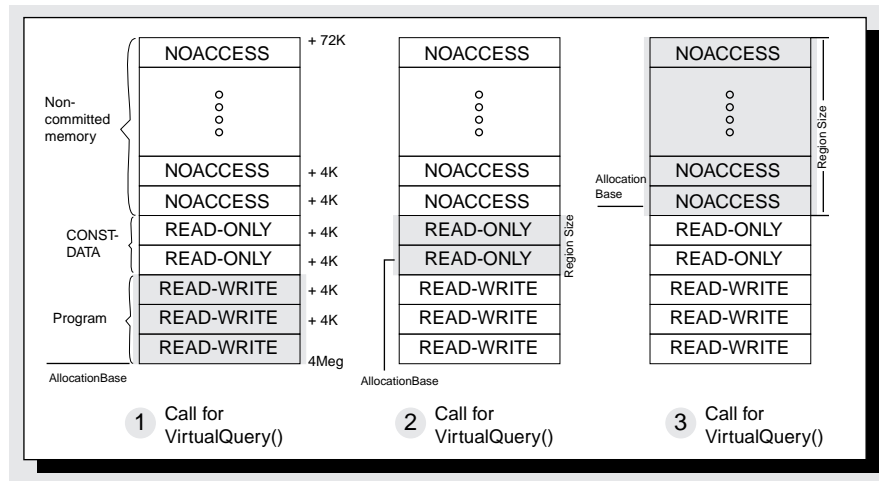
The `AllocationProtect` field contains the protection attributes for all the pages in the region in question. This is created in Windows 95 by one of the constants `PAGE_NOACCESS`, `PAGE_READWRITE` or `PAGE_READONLY`. The `RegionSize` field reports the size of the region so it can also indicate the number of pages with the same attribute. The `State` field indicates whether the pages in the region are free, reserved or committed. This is done by one of the constants `MEM_FREE`, `MEM_RESERVE` or `MEM_COMMIT`.

The last parameter indicates the page's source. This includes, for example, the following:

- Whether the pages were loaded from an EXE or a DLL file
- Whether they come from a data file that was copied to the memory
- Whether they are stored in the swap file

Unfortunately, unlike Windows NT, Windows 95 doesn't distinguish between the sources. It always returns `MEM_PRIVATE`, which indicates the source in the swap file.

Multiple calls of `VirtualQuery()`



A trick is available to determine the attributes of the pages in the region and for those of the following pages: Call the function several times in a row. Use the first page after the last call as the starting address each time. To do that, add the size of the region from `RegionSize` to the `BaseAddress` provided in `MEMORY_BASIC_INFORMATION`. You'll end up at the first byte of the subsequent page.

Protecting page attributes

Use the functions `VirtualProtect()` and `VirtualProtectEx()` to protect the attributes of a connected group of pages. Both functions expect the starting address of the region in the `lpAddress` parameter and its size in `dwSize`. All pages that contain at least one byte of the region so constituted will be included in the call.

```

BOOL VirtualProtect(
    LPVOID lpAddress,                // Starting address of area
    DWORD dwSize,                    // Size of area
    DWORD flNewProtect,               // New page attributes
    PDWORD lpflOldProtect            // Pointer to variable that gets old page
                                    // attribute
);

BOOL VirtualProtectEx(
    HANDLE hProcess,                 // Process whose memory is influenced
    LPVOID lpAddress,                // Starting address of area
    DWORD dwSize,                    // Size of area
    DWORD flNewProtect,               // New page attributes
    PDWORD lpflOldProtect            // Pointer to variable that receives old page
                                    // attribute
);

```

The attributes that the pages are to be given are included in the `flNewProtect` parameter. Again, one of the constants `PAGE_NOACCESS`, `PAGE_READWRITE` or `PAGE_READONLY` is used here. The last parameter is a pointer to a `LONG` variable where the function stores the previous attribute of the first page in the indicated region. You could also simply use `NULL` for this parameter.

Access tests

The ability to protect page attributes is not without its problems, of course. If something doesn't work as it was expected when protecting the attributes or when accessing the memory, the result is an exception error. That's why you should test whether you can access a particular address before accessing any memory. You may need to experiment to test. Five `IsBad...` functions will help with testing a given address for a particular function. The function result will tell you whether the address is suitable. For example, if `IsBadCodePtr()` returns `TRUE`, then the pointer that was used cannot be used to call a function. `IsBadReadPtr()` returns `TRUE` if the indicated memory is not allowed to be read.

Win32 functions for testing memory addresses		
Function	Result TRUE	Result FALSE
<code>ErglBadCodePtr()</code>	Cannot be executed	Can be executed
<code>IsBadReadPtr</code>	Cannot be read	Can be read
<code>IsBadWritePtr</code>	Cannot be written to	Can be written to
<code>IsBadHugeReadPtr</code>	Identical with <code>IsBadReadPtr()</code>	
<code>IsBadHugeWritePtr</code>	Identical with <code>IsBadWritePtr()</code>	
<code>IsBadStringPtr</code>	String cannot be read	String can be read

`IsBadRead` and `IsBadWritePtr()` both expect the starting address of the region to be tested and its length. The functions `IsBadHugeReadPtr()` and `IsBadHugeWritePtr()` are identical to these functions; they are from the 16-bit days and they were retained to make porting easier.

```

BOOL IsBadReadPtr(                // returns FALSE, if possible to read
    CONST VOID *lp,                // Starting address of area
    UINT ucb                       // Bytes in area
);

BOOL IsBadWritePtr(               // returns FALSE, if possible to write
    LPVOID lp,                    // Starting address of area
    UINT ucb                       // Bytes in area
);

BOOL IsBadHugeReadPtr(            // identical to IsBadReadPtr
    CONST VOID *lp,                // Starting address of area
    UINT ucb                       // Bytes in area
);

```



```

BOOL IsBadHugeWritePtr(                // identical to IsBadWritePtr
    LPVOID lp,                          // Starting address of area
    UINT ucb                            // Bytes in area
);

```

IsBadCodePtr() allows you to determine whether a given pointer can be used as a function pointer to call a function. If so, it will return FALSE.

```

BOOL IsBadCodePtr(                    // returns FALSE, if function call can be executed
    FARPROC lpfn                      // Function pointer
);

```

Finally, IsBadStringPtr() can be used to test access to C strings. The function searches through memory starting at the indicated address until it either finds a concluding NULL byte or until the maximum number of characters (as indicated in the parameter) have been run through. In either case, FALSE is returned if the memory can be read up to the particular character.

```

BOOL IsBadStringPtr(                 // returns FALSE, if string can be read
    LPCSTR lpsz,                     // Starting address of string
    UINT ucchMax                     // maximum number of characters
);

```

Bad Boys

We're using the name Bad Boys for a small console application. Bad Boys shows the use of IsBad functions in C. The program jumps into an endless loop within the starting function main() until the user pressed a key. Within this loop, a random address in the process's memory region is generated repeatedly. Then the different IsBad... functions are used to test whether and how the pointer can be used to access the memory. Appropriate messages are displayed on the screen as are the particular memory address.

You'll find the following program(s) on the companion CD-ROM



BADBOYS.C (C listing)

Querying the memory status

Many parameters of virtual memory management are preset in Windows 95. The page size is set at 4K and the granularity of VirtualAlloc() is set at 64K. Note this isn't true if you're working with Windows NT. If the application isn't compiled for Intel processors, you can get different page sizes and granularities depending on the target system. Therefore, Win32 API uses the GetSystemInfo() function to query these conditions. Win32 API also includes a function called GlobalMemoryStatus(). Use it to provide information about the status of the virtual memory and of the swap file.

Current memory status

When GlobalMemoryStatus() is called, the only parameter it expects is a pointer to a structure of the type MEMORYSTATUS. This is used to return the desired information to the calling routine.

```

VOID GlobalMemoryStatus(
    LPMEMORYSTATUS lpBuffer           // Pointer to MEMORYSTATUS structure
);

```

The first field in the structure has the name dwLength. It should be initialized by the calling routine with the size of the structure before GlobalMemoryStatus() is called. This way, the function learns if enough room is available in the structure, which is useful for future expansions.

```

typedef struct {
    DWORD dwLength;                // Size of structure in bytes
    DWORD dwMemoryLoad;            // current allocation state between 0 and 100
};

```

```

    DWORD dwTotalPhys;                // Size of physical memory in bytes
    DWORD dwAvailPhys;                // amount available
    DWORD dwTotalPageFile;           // Maximum size of swap file in bytes
    DWORD dwAvailPageFile;           // Free space in swap file
    DWORD dwTotalVirtual; // Total size of virtual memory of current process
    DWORD dwAvailVirtual;             // amount free
} MEMORYSTATUS, *LPMEMORYSTATUS;

```

This structure provides important information. This includes the size of RAM on the current machine, the size of the swap file, how much free space is in it, etc. The size of the virtual address space refers to the amount of memory that is available to the current process in its address space. In other words, this is the 4 Gigabyte, minus the space for the DLLs and the system code and the lower 4 Meg. The field `dwMemoryLoad` gives a percentage value between 0 and 100. This percentage estimates how much memory the system is using. A value of 100 means you're close to the edge and 0 means plenty of space is available.

Basics of memory management

The function `GetSystemInfo()` provides information about the basic parameters of memory management. It works based on the same principle as `GlobalMemoryStatus()`. It expects a pointer when it is called to a structure and provides the desired information in that structure. The following is a structure of the type `SYSTEM_INFO`.

```

VOID GetSystemInfo(

    LPSYSTEM_INFO lpSystemInfo        // Pointer to SYSTEM_INFO structure
);

```

Unlike `GlobalMemoryStatus()`, the length of the structure doesn't have to be initialized here. The function assumes enough memory is available.

```

typedef struct _SYSTEM_INFO {
    DWORD dwOemId;                // NULL
    DWORD dwPageSize;             // Bytes per page (4096)
    LPVOID lpMinimumApplicationAddress; // smallest addressable memory
                                     // location
    LPVOID lpMaximumApplicationAddress; // largest addressable memory
                                     // location
    DWORD dwActiveProcessorMask;   // only for NT
    DWORD dwNumberOfProcessors;   // Number of CPUs in system (Win95:1)
    DWORD dwProcessorType; // Processor type
                                     // Constant PROCESSOR_INTEL_386
                                     // Constant PROCESSOR_INTEL_486
                                     // Constant PROCESSOR_INTEL_PENTIUM
    DWORD dwAllocationGranularity; // Granularity for VirtualAlloc() - 64K
    WORD wProcessorLevel; // Processor type
                                     // 3 = 80386
                                     // 4 = 80486
                                     // 5 = Pentium
    WORD wProcessorRevision;
                                     // unknown
} SYSTEM_INFO, *LPSYSTEM_INFO;

```

Use these two functions for a fast summary of the current memory status. The following small program called `MEMSTAT.C` also shows this. It calls both functions, one after the other. Then it displays the different fields of the data structures on the console.

You'll find the following program(s) on the companion CD-ROM



MEMSTAT.C (C listing)

A question on performance

The last section described how committing previously reserved memory affects performance of a program. We'll explore this aspect in detail in this section. The first program in this section is called VIRALLOC.C. It provides specific answers to this question by implementing different strategies. The program is a console application. It simulates internally the workings of an application that stores more records in a continually increasing array. The array is reserved at the start of the program in its final size using VirtualAlloc(). However, no memory is committed yet. Commitment occurs only after new records are added, whereby three strategies used and their speeds are measured.

When calling the program from the command line, you'll determine how many records the program will generate, their size and which strategy is used when committing the memory.

```
virtualloc mode number size [lookahead]
```

You need to use one of the following identifiers for mode. It represents the different strategies for committing memory: COMPLETE, ONLYNEW or LOOKAHEAD. The second argument in the command line is the number of records to be generated. It can be a couple, or a couple of million. The size of the records is indicated in the third parameter. Filling the array can take considerable time. This depends on the number and size of the records you indicate. If that takes too long, interrupt the application at any time by pressing **Ctrl+Break**. If you selected the LOOKAHEAD strategy, you can use a fourth command line parameter. It contains the number of records that are to be previewed. You can quickly see what that's all about by looking at the program code.

VIRALLOC.C program code

The program starts with the main() function. Here it evaluates the command line parameters that are loaded in some global variables. For example, g_lMode uses one of the constants ALLOC_COMPLETE, ALLOC_ONLYNEW and ALLOC_LOOKAHEAD to create the desired commit strategy. Also, the record size is stored in g_lRecordSize. The record number is stored in g_lMaxRecords. Considering this input, an array of the appropriate size is allocated using VirtualAlloc(). The start pointer is stored in the global variable g_pRecords. No memory from the array is committed initially. That won't happen until main() is executed to the point where a program loop has been run through for each record that is to be created. The function PutRecord() is always called in this loop to store the next record in the array. PutRecord() determines the program's speed since this is where the different strategies are used.

PutRecords() is used to store the next array entry in memory and to make certain the associate memory space is committed. The way this is done depends on the strategy that was chosen. That's why a switch command separates the different strategies at the beginning of PutRecord(). In the simplest strategy, ALLOC_COMPLETE, VirtualAlloc() is called using the MEM_COMMIT flag so the entire memory, from the beginning of the array to the new record that is to be generated, is committed. So, you commit the entire part of the array that was used so far each time you call the function. This is the sledge hammer approach. Additional differences occur in the second strategy, ALLOC_ONLYNEW. Here, only the memory space for the new record is indicated at the MEM_COMMIT call to VirtualAlloc(). So, you make use of the fact that the entire previous memory space must already be committed and you just concentrate on the new space.

The third solution is even better. It looks ahead and always commits memory for a large number of array elements all at the same time. If the call routine uses the LookAhead parameter in the command line, this value will be used for the number. Otherwise, the default of 100 is used. This strategy has one big advantage. A VirtualAlloc() call doesn't have to follow every call of PutRecord(). Instead it only appears when all array slots that were committed by the look ahead process are used. Then no more calls have to be made for Preview-1.

You'll naturally waste some memory doing this if memory was committed for some records at the end of the array but it wasn't ever written to. Fortunately, that's not a big problem. After all, you don't have to set the number of records to be looked ahead so large that you waste hundreds of kilobytes. We'll show you how to work with the three strategies in actual practice shortly. Considering the information in the table on the right, the third method (Lookahead method) is the

Number of Records	1000	10000	100000
Size in Bytes	2000	200	20
COMPLETE in Sec.	0:550	2:937	20:940
ONLYNEW in Sec.	0:207	0:808	7:317
LOOKAHEAD in Sec.	0:149	0:406	0:546

fastest. These measurements were performed on a Pentium 90 with 16 Meg RAM. An array 2 Meg in size in each case was to be written. In the first case, it contained 1000 records that were 2000 bytes in size. In the second case, 10,000 records were 200 bytes in size. In the third case, 100,000 records were 20 bytes in size.

Obviously the process requires more time as the number of records increases. You may be, however, surprised the Lookahead method is the fastest and the Complete method is the slowest. The more records you add, the greater is the difference between the different strategies. The Lookahead method with 1000 records is only three times as fast as the Complete method. However, the speeds are different by a factor of 40 when the Lookahead method has 100,000 records.

So the program shows that it makes sense to consider committing memory within your programs. Depending on the situation, consider using Lookahead to achieve definite increases in speed.

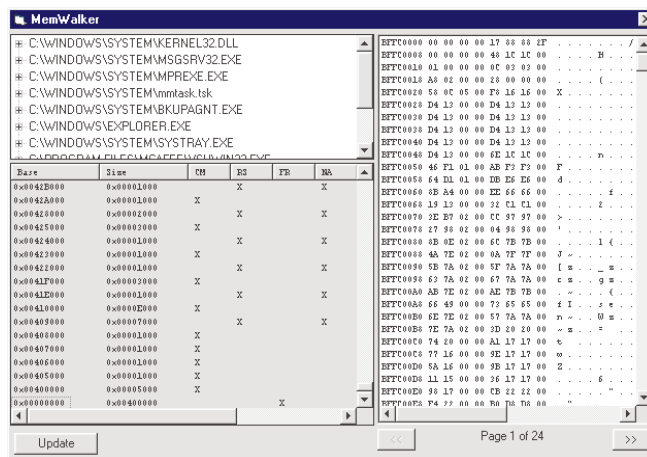
A memory monitor

A memory map can provide a visual representation of the processes and data stored in allocated memory. This is the purpose of the MemWalker program that you'll find on the companion CD-ROM. Most of the program was developed using Visual Basic 4.0 (the 32-bit version). Therefore, you need Visual Basic 4, or at least its runtime DLLs, to use MemWalker. The VB modules are supported by a MEMDUMP.DLL. This DLL, created in C, is used for the physical reading of memory. We'll talk more about this later.

Using MemWalker

The following illustration shows MemWalker at work. The upper left corner of the application window contains a TreeView control. This is one of the new common controls of Windows 95. We use it to display the processes currently in execution. In other words, it shows you which "programs" are active.

A look at the Explorer's memory with MemWalker



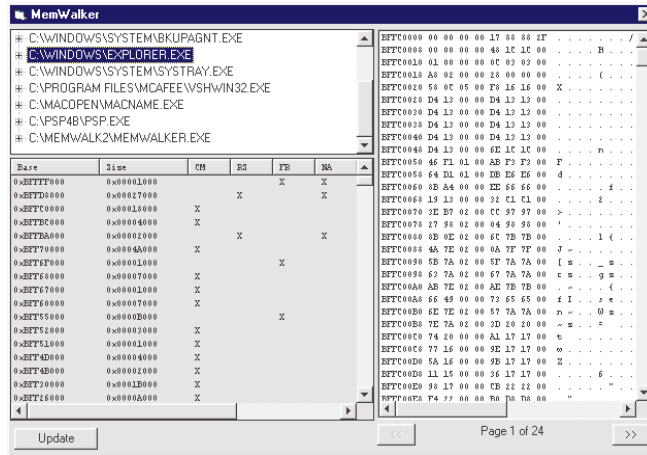
The individual entries of the TreeView can be expanded to show more information about each process. This includes the names and base addresses of modules loaded within the process. A module can be a DLL or EXE file that the program loader loaded when the process was started, based on references in the file. Also, the application itself may have loaded some modules using the API functions LoadLibrary() or LoadModule() during execution. There is no indication of how the modules became located in memory. All you can see is how many there are and the file names. There may be a considerable number, as for the 32-bit EXE file of VB4 shown below. What you see here is only a small portion of all the modules loaded. However, you can easily use MemWalker to view the entire list.

You'll find the following program(s) on the companion CD-ROM



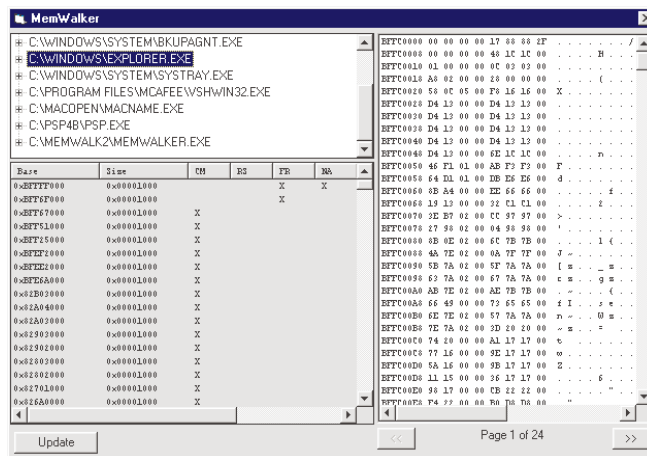
VIRTALLOC.C (C listing)

By clicking a process, you make its modules visible. This is just a small part of the DLLs and other components loaded by the 32-bit version of VB4.



Click a new process in TreeView to change the memory map in the underlying window. This feature uses another of the new common controls, specifically a ListView control in Report view. The memory map shows a view of memory comparable to what you get with VirtualQueryEx() when you search a process's address space for pages with common attributes. The first column shows the region's base address in memory. The size of the region's base address appears to the right. You will notice that the three low-order hex digits of the size are always zeros. Since 0x1000 in hexadecimal is 4K, we can conclude that the sizes are always multiples of 4K. Each region shown consists of an integral number of 4K blocks called pages.

Clicking the column header sorts the memory map display (here the reserved regions are sorted by size)



MemWalker searches only the first 3 Gigabytes of memory for a process, however. The upper 1 Gigabyte is reserved for system use and cannot be accessed. To view the contents of a memory region, click the desired line in the 'Base' column. The listbox to the right will display the contents with eight bytes per line, in the following order:

1. The base address for the line
2. The eight bytes in hex
3. The eight bytes in ANSI (for all displayable characters).

This format allows you to easily spot regions with ANSI strings. If a selected region has more than one page, the two buttons below the listbox become active. Use these buttons to scroll through the region page by page. However, not every region that

appears in the list can be displayed. The flags indicated to the right of the size column are checked for certain settings. The following table explains their meaning:

For the contents to be displayed in the listbox, the memory must at least be committed; otherwise, you cannot access it. To sort the regions by attribute, click the corresponding column header. For example, click 'CM' to bring all the committed regions to the top. In the same way, you can order the list by descending 'Size' to see the largest regions first.

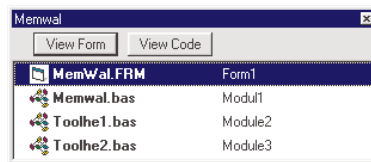
Finally, we need to talk about the 'Allocation' column. Although MemWalker does not identify who allocated a particular memory region, you can at least tell which regions were reserved by the same caller using VirtualAlloc(). Such regions will have the same address in the allocation column. Most of the addresses in this column begin on a 64K boundary and are simply the addresses returned by VirtualAlloc(). Sometimes you'll see a region whose allocation base starts on an address not divisible by 64K, but only beyond 0x80000000. This is where the system managed memory begins. Obviously the system allows itself a somewhat finer granularity here than is permitted for executing processes.

Flag	Meaning
CM	Memory is committed
RS	Memory is reserved but not committed
FR	Memory is free
NA	Memory is not available
RW	Memory can be read and written
RO	Memory can only be read

MemWalker program code

Although MemWalker in its present form is a helpful tool, you can customize it to your needs. The source code and project files on the CD-ROM allow you to do this quite easily. The program is based on the VB Make-file called MEMWAL.MAK, shown below.

*Files in the
MEMWAL.MAK
project*

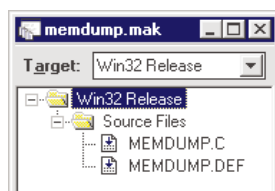


First, you see the form MEMWAL.MAK. It contains the program's various controls and their related code. This is responsible only for controlling interaction with the user. It is not concerned with finding the names of the executing processes nor the memory region attributes or contents. These tasks are handled by three VB modules: MEMAL1.BAS, TOOLHE1.BAS and TOOLHE2.BAS.

The following is the program code for the MEMWALKER form. It reveals much about the program's internal functioning. The hardest task is determining which processes are executing and which modules are loaded within them. This involves checking system information that is not readily accessible. Fortunately, KERNEL32.DLL includes several API functions designed for this purpose. These functions are called the 'ToolHelp' functions. Call these functions from the TOOLHE2.BAS module within MemWalker. Declarations required for the call are found in TOOLHE1.BAS. The next two listings show how to use the Toolhelp functions under Visual Basic.

MEMWAL.BAS is next. The memory region for a process is scanned here using VirtualQueryEx(). Use the procedure MemWalk for this purpose. It expects the handle of the current process and a ListView control when called. The procedure fills this control with data for the individual memory regions. It assumes that the columns have already been initialized with the proper sequence, widths and headings. Another procedure within the module is SHOWMEM. It displays the contents of a page. To do this, it accesses MEMDUMP.DLL, which loads the desired page contents into the program's listbox.

*Files in the
MEMDUMP.MAK
project*



As this illustration shows, MEMDUMP.DLL of the required DEF file consists simply of a C module named MEMDUMP.C. The first function in the MemDump is MemDump(). Its job it is to reproduce the page contents in the listbox. It reads the contents of the desired page using the function ReadProcessMemory() from the Win32 API.

A string is formed from the individual lines using printf(), SendMessage() and LB_ADDSTRING. The string is added to the end of the listbox.

Also, the listing of MEMDUMP.C shows uAdd, which is another important function. It adds two long values. Since long values must always be signed in Visual Basic, this is an unexpectedly difficult task. Once we reach the value 0x80000000 in computing memory addresses, we are in the range of negative numbers. This, of course, makes no sense. Calling the uAdd function eliminates the problem of addresses over 2 Gigabytes suddenly turning negative.

Heaps

Use the different Heap functions when you want to manage small blocks instead of large blocks of memory. Then you won't need to worry about whether memory has been committed. Heap functions are used to perform heaps similar to those provided by the standard C functions malloc(), realloc() and free(). These routines are based in the C runtime library on the heap functions of the Win32 API. So, if you're familiar with those functions and you don't depend on the API's advantage of managing several separate heaps, don't use the API functions.

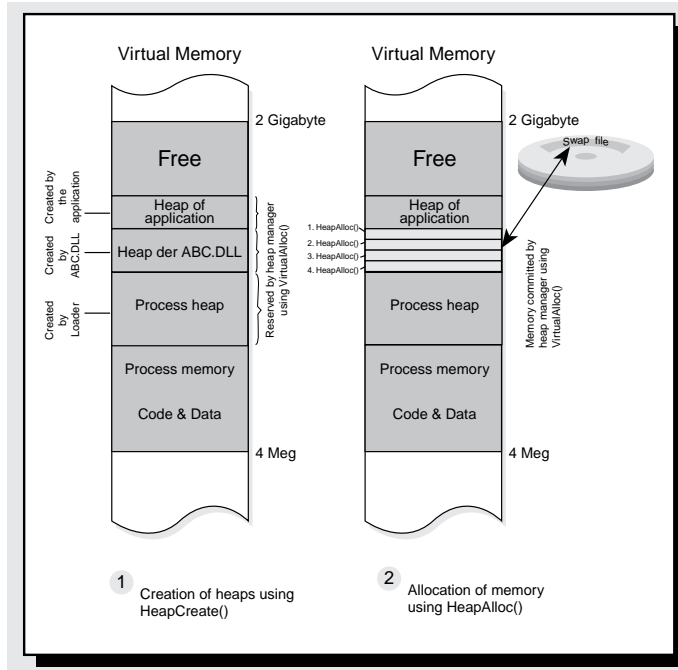
You'll find the following program(s) on the companion CD-ROM



MEMWAL.FRM (VB listing)
TOOLHE1.BAS
TOOLHE2.BAS
MEMWAL.BAS
MEMDUMP.C (C listing)

Win32 functions for heap management	
Function	Task
HeapCreate	Creates a new heap
HeapAlloc	Allocates memory from a heap
HeapFree	Frees up a block of memory previously allocated by a heap
HeapReAlloc	Expands or reduces a block previously allocated by a heap
HeapDestroy	Deletes a previously generated heap
HeapSize	Provides the size of a previously allocated block.
GetProcessHeap	Provides the handle to the heap of the current process
GetProcessHeaps	Fills an array with the handles of all heaps of the current process

Using Win32 heaps



It's useful to create several heaps using the Heap API everywhere that objects are to be stored in vastly differing sizes using heaps. Depending on the relative sizes of the objects and how often you delete them and generate new ones, if use them jointly you will have more fragmentation problems than if you split up the objects in separate heaps. The memory for heaps is drawn from the application's virtual address space and is allocated internal with VirtualAlloc(). This way you can set up small heaps and large heaps.

Heaps are created using HeapCreate(). A heap handle is returned. It must be used when calling heap functions later to identify the addressed heap. HeapAlloc() allocates regions of memory, HeapRealloc() changes their size, and HeapFree() frees them up again. The semantics are noticeably reminiscent of the standard C library.

Process heap

The process heap exists at the very start of the process. The program loader generates it when the process is loaded. It uses the heap to store information that the system kernel must manage for the particular process in its section of memory. Nobody prohibits the process from allocating its memory using this heap. All you need for that is the process heap's handle, which is provided by GetProcessHeap() as a functional result.

```
HANDLE GetProcessHeap( VOID );
```

Use the GetProcessHeaps() function to determine the handles for all the heaps that currently exist in a process. Assign it a pointer to an array with PHANDLES (they're DWORDs or longer). During this time it writes the handle of the existing heaps. It also expects the number of free elements kept in it at the same time. By doing so, it doesn't write out beyond the buffer.

```
DWORD GetProcessHeaps(
    DWORD    NumberOfHeaps,           // Number of items in array
    PHANDLE  ProcessHeaps            // Pointer to array
);
```

The DWORD that is returned tells you how many heaps are present. If more heaps are present than you have prepared for in the array, no entry is written to the array. In this case, you should enlarge the array accordingly and call the function again, indicating the correct number.

Generating a New Heap

If you don't want to use one of the existing heaps, use HeapCreate() to create your own.

```
HANDLE HeapCreate(
    DWORD    fOptions,                // Options
    DWORD    dwInitialSize,           // Initial size
    DWORD    dwMaximumSize            // Maximum size
);
```

Along with the a flag, HeapCreate() expects the initial size of the heap or the number of bytes that should be made available. The figure is rounded off to sizes of pages. A corresponding number of pages is immediately committed. To get the maximum size, input NULL if you don't want to intentionally limit the size. By doing so, the heap will automatically grow with the increasing memory needs. You can also indicate a maximum number in bytes. We cannot recommend this approach, however, because then any attempt to allocate memory beyond the maximum size with HeapAlloc() will fail.

The first parameter, fOptions, is used to prescribe the behavior of the heap in certain situations. Two options are possible, and, with them, two flags available that can be combined:

HEAP_GENERATE_EXCEPTIONS

When this flag is used, no memory is committed. Instead of returning NULL when HeapAlloc() is called, an exception is generated. This option is usually designed more for system applications.

HEAP_NO_SERIALIZE

Several threads of a process can, in certain situations, access a heap at the same time using different Heap functions. Therefore, the access within the different Heap functions is serialized by default. Only one thread at a time can access a particular Heap function using one of the Heap functions. All the others that are addressing the same heap have to wait until the first one is done. If a process doesn't work with multiple threads, you can bypass this serialization. This saves a little time. If you want to take advantage of this, use this flag when you create a heap.

Allocating Memory from a Heap

If HeapCreate() did not return a NULL (for error), the result of the function provides you with the handle of your new header. Heap memory can then be requested immediately using HeapAlloc().

```
LPVOID HeapAlloc(
    HANDLE   hHeap,                // Handle of heap
    DWORD    dwFlags,              // Flags, determine the execution
    DWORD    dwBytes               // Number of bytes to reserve
);
```

Most of the Heap functions expect the heap's handle as the first argument with which they are to work. This is also true for HeapAlloc(). Flags are used as the second parameter. This is therefore where the two flags we already discussed can be placed. These flags are HEAP_GENERATE_EXCEPTIONS and HEAP_NO_SERIALIZE. However, that only makes sense if you want to switch off the presetting for this call that was established using HeapCreate(). The flag HEAP_ZERO_MEMORY is practical; it automatically fills all the bytes in the allocated block of memory with 0. This is an easy way to start at a clearly defined state when you are building new data structures. The third parameter must contain the size of the desired buffer as a number of bytes.

If the function returns NULL, no memory could be allocated in the desired amount, although this should not occur frequently. As a rule, after you call the function you have a pointer that points to the start of the reserved block of memory. The region is not only reserved as a result of the call but is also committed. So, you can use the pointer that is returned to directly access the region. If you want to subsequently reduce or enlarge the block, use the HeapRealloc() function:

```
LPVOID HeapReAlloc(
    HANDLE   hHeap,                // Handle of heap
    DWORD    dwFlags,              // Flags, determine the execution
    LPVOID   lpMem,                // Pointer to area
    DWORD    dwBytes               // New size in bytes
);
```

Besides the handle of each heap, the starting address of the region and its new size in bytes are expected here. Again, flags can be HEAP_GENERATE_EXCEPTIONS, HEAP_NO_SERIALIZE or HEAP_ZERO_MEMORY. If you're expanding a region and use HEAP_ZERO_MEMORY, a message appears saying the added portion will be automatically filled with zeros. Finally, the function also recognizes the constant HEAP_REALLOC_IN_PLACE_ONLY. This means the following: Sometimes when you enlarge a region of memory it must be moved because the memory space following the end of the region has already been committed elsewhere. The block cannot be changed from where it is now. All you can do is look for the next free region on the heap that is large enough to accept the region in its new size. If such a region is found, the contents so far are copied into it. A pointer to the new region is returned to the calling routine as a result of the function. The old region is freed and can be reserved during the next call.

However, if you input HEAP_REALLOC_IN_PLACE_ONLY, the region will not be moved if it can't be expanded where it is now. In this case the function returns a NULL. In all other cases, the function result that is returned is the new address of the region. This may be very different from the old one. So you need to make sure you update all pointers within a program that point to this region.

If you want to determine the size of the block of memory in the meantime, use function HeapSize(). The flags don't play any remarkable role when this function is called, and you're already familiar with the other parameters.

```

DWORD HeapFree(
    HANDLE   hHeap,                // Handle of heap
    DWORD    dwFlags,              // Flags, in this case NULL
    LPCVOID  lpMem                 // Pointer to area to be released
);

```

What you get back is the length of the region of memory in bytes or 0xFFFFFFFF, if you input an unknown heap or block of memory. Once you have decided to free a block of memory, invoke HeapFree():

```

BOOL HeapFree(
    HANDLE   hHeap,                // Handle of Heap
    DWORD    dwFlags,              // Flags, here NULL
    LPVOID   lpMem                 // Pointer to area to be released
);

```

If the region indicated was able to be freed, you'll receive the result TRUE. Afterward, the pointer to the region cannot be used anymore to access the memory. One way to remove all memory regions of a heap is to call HeapDestroy(). This destroys the entire heap. The heap handle becomes invalid but an application is free to call HeapCreate() again and recreate a heap.

```

BOOL HeapDestroy(
    HANDLE   hHeap                 // Handle of heap to be destroyed
);

```

Memory operations

The Windows API includes four functions that you can use to execute elementary operations with memory blocks: copy, fill, move and delete the contents.

However, you may be surprised if you look for the definition of these functions and the DLL containing them. These aren't even API functions because the presumed API calls are converted by different macro definitions into calls to the C compiler's runtime library. First, the presumed API functions are mapped to different function names in WINBASE.H:

Functions for working with blocks of memory	
Function	Task
CopyMemory	Copies a region of memory
FillMemory	Fills a region of memory with a constant byte
MoveMemory	Moves a block of memory
ZeroMemory	Fills a region of memory with zeroes

```

// WINBASE.H

// ...
// ...
// ...

#define MoveMemory    RtlMoveMemory
#define CopyMemory    RtlCopyMemory
#define FillMemory    RtlFillMemory
#define ZeroMemory    RtlZeroMemory

```

The new function names in WINNT.H are then mapped to the known functions from the runtime library. Then it doesn't make a difference any more if use the standardized functions from the runtime library immediately.

```

// WINNT.H

// ...
// ...
// ...

```

```

#if defined(_M_IX86) || defined(_M_MR000) || defined(_M_ALPHA)
#define RtlMoveMemory(Destination,Source,Length) memmove((Destination),
                                                         (Source),(Length))
#define RtlCopyMemory(Destination,Source,Length) memcpy((Destination),
                                                         (Source),(Length))
#define RtlFillMemory(Destination,Length,Fill) memset((Destination),(Fill),
                                                         (Length))
#define RtlZeroMemory(Destination,Length) memset((Destination),0,(Length))
// ...
// ...
// ...

#endif

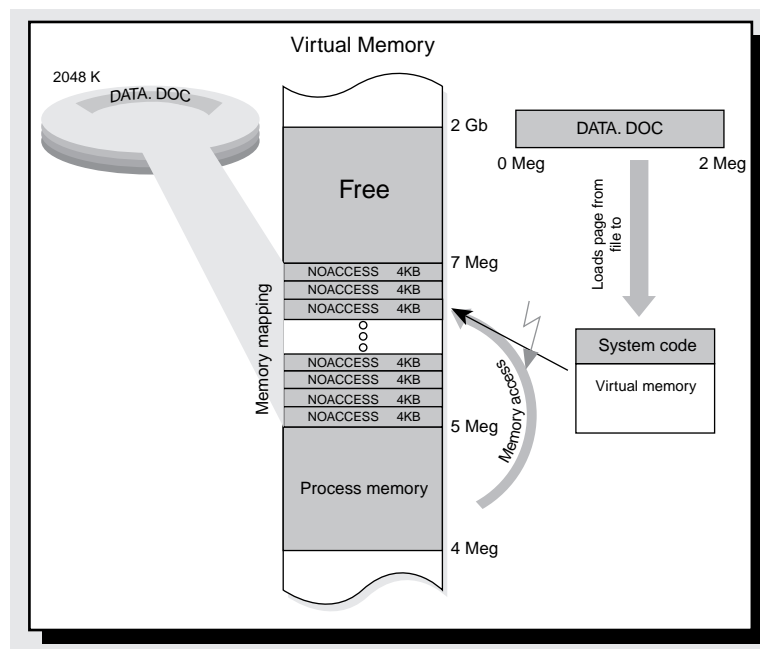
```

Memory Mapped Files

We've talked about memory mapped files (MMF) as one way to save room in the swap file in that you map EXE files and DLLs directly to the memory. Thus, the corresponding pages are not created in the swap file. Instead, they're drawn from the real file on the disk. Whenever one of these pages is later loaded, it is accessed directly from the program file. This not only saves memory but also means the file doesn't have to be loaded into the swap before your pages can be loaded from there. Of course that helps your performance.

Now, not only is the potential of the memory mapped files available to the system code, but it can also be used by user code as a few functions from the Win32 API. Here we see another very unexpected aspect of the MMF principle. Working with files within a program has been revolutionized: you will never again need to make a single call to `Seek()`, `Read()` and `Write()`. Instead, we use the API to map a file to memory and then we use pointers to access it as we would if it were completely in memory. We also avoid manipulations if several sections of a file are supposed to be committed in memory at the same time. If you've created routines for processing very complex file formats, you'll appreciate the MMF idea. Much of the effort has been eliminated and the rest is a lot clearer and easier with which to deal.

Accessing files without `Seek()`, `Read()` and `Write()` thanks to memory mapped files



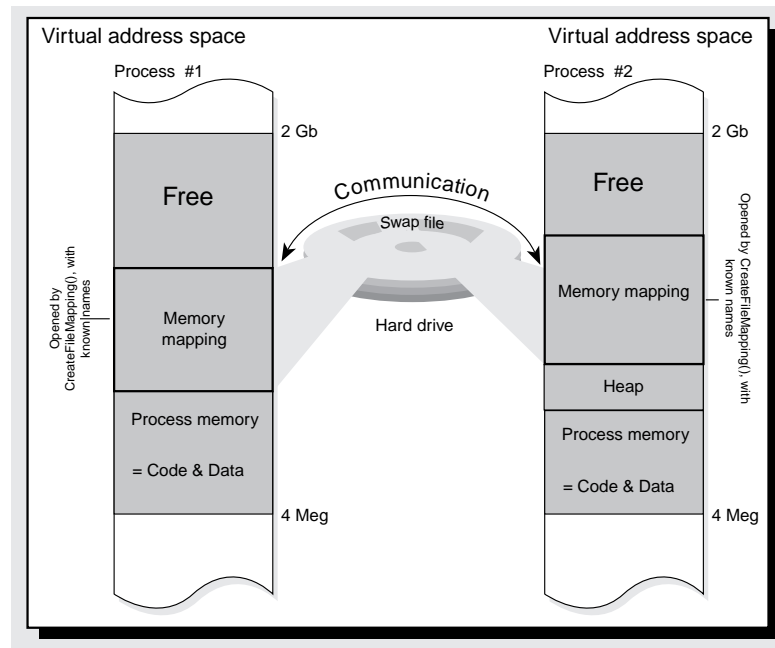
With every access to the region to which the file is mapped, the virtual memory manager automatically loads the corresponding page from the file into memory if it hasn't already been loaded. An intelligent manager makes certain pages that were written to are not written back to the file immediately after every write access. Instead it will wait for additional accessions. This is called *lazy write*.

Interprocess communication

MMF is a multifaceted idea if you consider the memory mapped files are the only way for processes to manage larger amounts of data with other processes under the Win32 API. This is a challenge alone in view of the sealed address spaces of processes under Windows 95. Memory mapped files show you how to escape out of this tangled web. They can be used to map a file to multiple address spaces at the same time. Several processes participate in dealing with an MMF object by giving it a name that is known to all the participants. After that, each process just has to open the MMF object using this name to gain access. Each write access to the address region where the file is mapped to appears immediately in the address spaces of all participating processes. This is because the virtual memory manager inserts one and the same pages in all address spaces. So the same thing happens here that occurs when mapping several copies of DLLs or processes in different process memories.

Also you don't even have to have one of the participating processes provides a file in which the jointly used memory is swapped out. Instead of providing a file name when you open an MMF object, the file will be pulled from the swap memory if you input 0xFFFFFFFF.

Interprocess communication using memory mapped files



Win32 functions for working with memory mapped files	
Function	Task
CreateFileMapping	Creates a Memory Mapped File object (MMF object) for a given file
OpenFileMapping	Opens an existing MMF object
MapViewOfFile	Maps part of the file to a segment in memory
MapViewOfFileEx	Expanded variant of MapViewOfFile() and indicating the desired starting address of the map
UnMapViewOfFile	Unmaps a segment of the file from memory
FlushViewOfFile	Writes all pages that were changed but not yet saved, to the file so the changes can't be lost

Creating a memory map

The “fun” begins with creating a memory map. The first step uses the standard `CreateFile()` function of the Win32 API to open the file that is to be mapped. That’s how you get the file handle that you need to build an MMF object. Also, this protects the file from being opened by other processes. Otherwise, this would lead to problems during file accession (key word: race conditions).

When you call `CreateFile()`, it expects flags that tell it how you want to be able to access the file. If you want read and write permission, input `GENERIC_READ + GENERIC_WRITE`. If you just want read permission, don’t use `GENERIC_WRITE`. By using the file handle that is returned, you can generate an MMF object using `CreateFileMapping()`.

```
HANDLE CreateFileMapping(
    HANDLE hFile,                // File handle
    LPSECURITY_ATTRIBUTES lpFileMappingAttributes, // Win95: NULL
    DWORD flProtect,             // Page attributes
    DWORD dwMaximumSizeHigh,     // Maximum size of file, Lo-DWORD of QWORD
    DWORD dwMaximumSizeLow,      // ditto, Hi-DWORD of QWORD
    LPCSTR lpName                // Pointer to string with name of MMF object
);
```

The handle of the file needs to be the first parameter. As we mentioned, if the memory is to be taken from the swap file, you need to input `0xFFFFFFFF`. Give permission attributes in the second parameter to the MMF object at the same time you create it. Note, however, that you can only do that under Windows NT for now. Simply input `NULL` here. Use the `flProtect` parameter to define the attributes of pages that will store the contents of the file and map it to the address spaces of the different processes. As a rule, you will use `PAGE_READWRITE` here so you can read and write to the pages. You would only use `PAGE_READONLY` with program code or read-only data. In this case, an attempted write access would generate an exception.

You’re probably already guessing that even the 32-bit era won’t last forever. The next era could be a 64-bit integer, a quad word (4 x 16 bits), for which there isn’t even a standard type identifier yet in C. So, two separate `DWORD`s provide the maximum file size, although you probably won’t need the high `DWORD` in `dwMaximumSizeHigh` at first. After all, the content of `dwMaximumSizeLow` is enough to increase files up to a size of 4 Gigabyte.

If an existing file is only to be read, you can just input a 0 for the size. In this case, the MMF object that is generated represents the existing file, but it can’t grow any larger. A size that you input is considered to be the maximum size. In other words, a previously existing file can be expanded by accessing memory after the previous end (up to the size indicated here).

The last parameter is a pointer to a C string that can be used to give the new MMF object a name. If the object will only be used within a process, the name usually is not needed so you can simply input `NULL`. The names have to follow the same rules as for names of mutexes, semaphores and events. The name can include all symbols except for backslash (`\`), and can’t be longer than 64 characters. It should be unambiguous so you can’t easily conflict with other processes that might have the same name.

Once the desired MMF object has been generated, `CreateFileMapping()` will return a valid handle rather than `NULL` for the new MMF object. This is also true if you have used a name and the named MMF object already exists. If several processes are communicating using an MMF object, each one opens it when it starts using `CreateFileMapping()` by indicating the appropriate name. The object will be generated if it doesn’t exist. Otherwise, if it does exist, you’ll receive a handle for the already existing object. In either case, the call has done its job.

Accessing existing MMF objects

However, if you want to know exactly, you can use another command, like `OpenFileMapping()`. Then the call is only successful if the indicated MMF object already exists. So, by using a test call to `OpenFileMapping()` you can verify whether a certain MMF object already exists or if it must be generated first.

```

HANDLE OpenFileMapping(
    DWORD    dwDesiredAccess,           // Access flag
    BOOL     bInheritHandle,           // Inheritance flag
    LPCSTR   lpName                     // Pointer to name
);

```

Use the parameter `dwDesiredAccess` to indicate the desired access to the file's contents. If you only want to read it or only write to it, or use both types of access, these options are indicated by the three predefined constants `FILE_MAP_WRITE`, `FILE_MAP_READ` and `FILE_MAP_ALL_ACCESS`.

Use the second parameter to specify whether the returned handle should be inheritable by daughter processes. Input `FALSE` if your process doesn't generate any new (daughter) processes with which it will communicate using the MMF object when it executes. Finally, the last parameter expects a string pointer that points to a C string with the name of the MMF object. It has to contain the name by which a previous call routine to `CreateFileMapping()` generated the MMF object.

The function then looks in the name memory to see if this MMF object already exists. If not, it returns `NULL`. Otherwise, it returns a handle that can be used to access the MMF object when calling other API functions. From this moment, the calling process interacts with the underlying file. It can map it in its memory to access it.

Mapping to the memory

Use the `MapViewOfFile()` function. It expects the MMF handle for the first parameter. The desired access mode to the data contents is expected as the second parameter. As with `OpenFileMapping()`, use `FILE_MAP_WRITE`, `FILE_MAP_READ` or `FILE_MAP_ALL_ACCESS` here.

```

LPVOID MapViewOfFile(
    HANDLE    hFileMappingObject,       // Handle of Mmf object
    DWORD     dwDesiredAccess,          // desired access to file
    DWORD     dwFileOffsetHigh,         // Start offset in file, Lo-DWORD of QWORD
    DWORD     dwFileOffsetLow,          // ditto, Hi-DWORD of QWORD
    DWORD     dwNumberOfBytesToMap      // Size of area to map in bytes
);

```

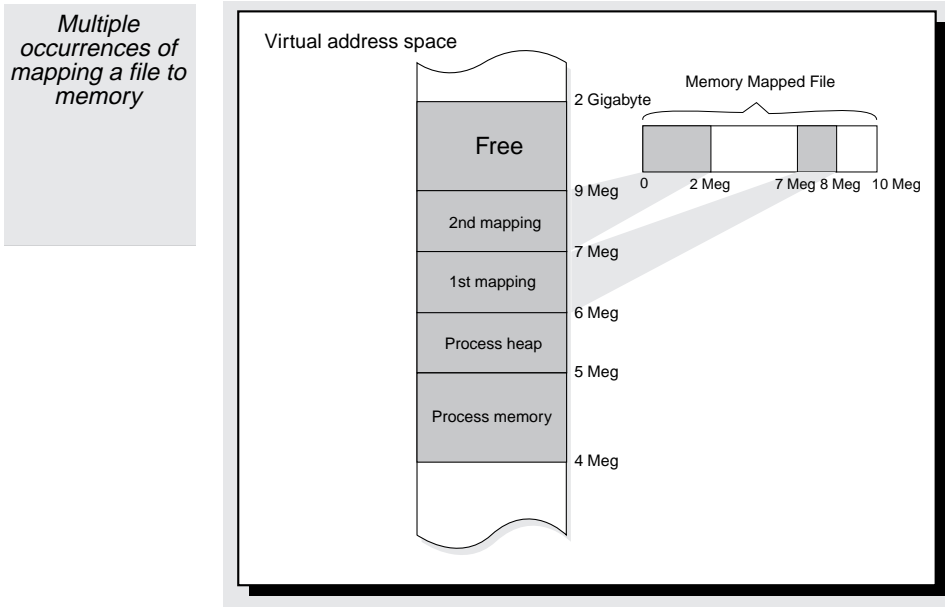
The offset within the file is next. It starts from the contents of the file that are to be mapped to the memory. Again, this is as a 64 bit integer, and therefore it is distributed over two DWORDs named `dwFileOffsetHigh` and `dwFileOffsetLow`. The last item is the number of bytes that are to be mapped. With the starting address and the number of bytes, you have specified the region of the file that is to appear in memory. This also specifies the size of the virtual memory the function is to reserve so the contents of the file can fit there.

There is an important limitation for the starting address, however. It has to be a multiple of the allocation granularity. That is the 64K region under Windows 95 that was mentioned with `VirtualAlloc()`. So in `dwFileOffsetLow`, you can use null, 0x00010000 (64K), 0x00020000 (128K), etc. for the offset. Do not, however, use any addresses between the addresses. So the low word of `dwFileOffsetLow` always has to be 0. This limitation is connected with the fact that `MapViewOfFile()` depends on `VirtualAlloc()` for the mapping.

This has some interesting implications for mapping EXE and DLL files.

The pointer returned by `MapViewOfFile()` represents the start of the region in the address space of the calling process to which the indicated file region was mapped. After this function has been called successfully, you can access the file using the pointer. However, after the function returns, you should always check to see if the pointer contains `NULL`, indicating an error. Besides an erroneous parameter when the function is called, the function can fail in Windows 95 if this means that space has to be created in the swap file and the swap file can't be made any larger.

When several processes map to the same regions of a file using an MMF object, they all obtain common access to it. Also, a process is permitted to call the `MapViewOfFile()` function several times to map different regions of a file to separate regions in memory. However, the simplest thing to do is to use `MapViewOfFile()` to map the entire file to memory all at once.



However, it is important that the last byte in the region to be mapped doesn't exceed the maximum size that was indicated with `CreateFileMapping()`. If it does exceed the previous end of the file, the file will be enlarged accordingly when it is accessed.

`MapViewOfFileEx()` is an expanded version of `MapViewOfFile()`. The parameters are the same for both functions. However, you can define the starting address in `MapViewOfFileEx()` for the mapping in the address space of the calling routine. That usually won't help normal user code much. However, the program loader uses this ability to have program code from a mapped EXE file start at a defined location in the address space. Under Windows 95 this is at the address `0x0400000` (4 Meg).

```
LPVOID MapViewOfFileEx(
    HANDLE   hFileMappingObject,
    DWORD    dwDesiredAccess,
    DWORD    dwFileOffsetHigh,
    DWORD    dwFileOffsetLow,
    DWORD    dwNumberOfBytesToMap,
    LPVOID   lpBaseAddress    // only this parameter, different with desired
                                // base address
);
```

If you write to the file using its memory map, the changes won't be written immediately. This avoids a costly I/O operation every time a write access occurs. So, situations might arise where you will want to force the data to be written to avoid any loss of data. You're also certain the changes have been saved on the hard drive. The `FlushViewOfFile()` function allows you to do this.

```
BOOL FlushViewOfFile(
    LPCVOID   lpBaseAddress,           // Starting address of area
    DWORD    dwNumberOfBytesToFlush   // Number of bytes to flush
);
```

The starting address that you give the function doesn't have to agree with the start of the map in memory. The function determines by itself which page the region begins in and it writes the changes to disk. The number of successive pages that will need to be written depends on the size of the region. This is specified by the parameter `dwNumberOfBytesToFlush`. If use 0 here, the entire memory to the end of the respective mapping is written to disk. Any other value lets the function calculate the page on which the memory block ends. All pages, including this page, are written to disk.

Unmapping a mapping

If you want to unmap something, use `UnmapViewOfFile()`. After you have successfully called this function, any other attempts to access the memory in the array previously reserved for this mapping generate an exception.

```
BOOL UnmapViewOfFile(
    LPCVOID lpBaseAddress    // Starting address of mapping to be removed
);
```

The best way to return MMF object itself it to use the `CloseHandle()` function and the object is deleted. Then you can use `FileClose()` to close the associated file.

The PE (Portable Executable) Format

We showed earlier the format of the executable files on the hard drive must meet certain criteria so it doesn't interfere with virtual memory management. Most importantly, it must guarantee the program code and its data are mapped to memory so EXE and DLL files can be used concurrently by multiple processes. The answer to this challenge is the Portable Executable format, or PE for short. When the Windows 95 program loader receives a request to start an application, it interprets the contents of a PE file. It then uses the contents to build the program's image. This is in the same way the linker expects it when building a PE file.

This section talks about the basic structure of PE files and the ideas behind this structure. You'll see that PE files contain a series of other elements required for execution besides the actual code and data from an application. We'll look closely at the information about imported and exported function. A sample program at the end of this section shows how to read and view this information.

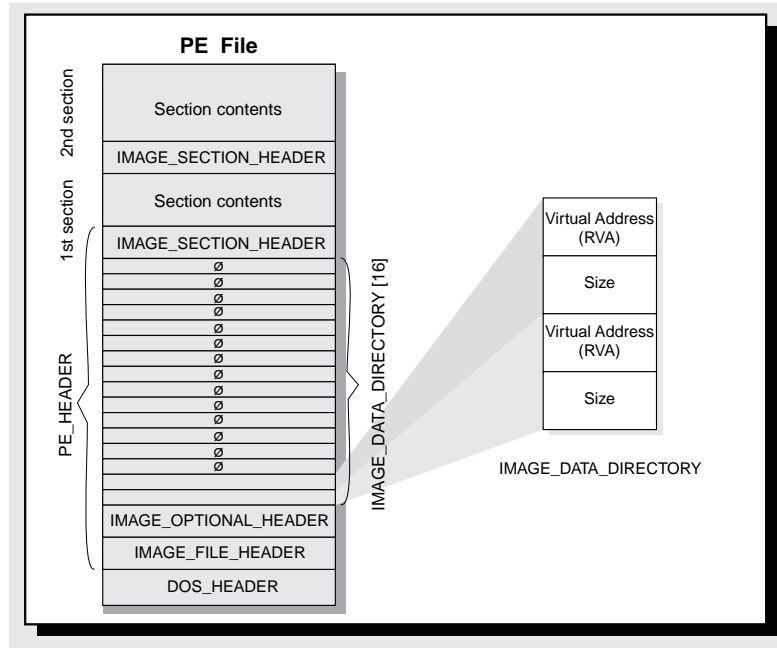
PE file structure

The Win32 SDK for Windows doesn't officially document the PE formation for executable EXE files and DLLs. On the Win32 SDK CD-ROMs for Windows NT, a file `PE.TXT` was located (at least was for awhile) in the `\DOC\FORMAT` directory. Microsoft provides a general description of the PE format for interested developers in this file. Unfortunately, some essential details for the concrete interpretation of an existing PE file under Windows 95 are not included. A series of types and structures that help here are defined in the `WINNT.H` include file. They reproduce the concrete structure of the individual elements of a PE file. Unfortunately, the description of the interaction among these elements is not quite long enough, so you still have to experiment using an actual example. You'll eventually recognize the basic structure of PE files.

The following illustration shows that a PE file contains different sections that serve different tasks. The familiar DOS header is at the beginning. It's followed by the new PE header and finally a series of different sections that are managed through a section table. These sections are called segments in DOS.

These sections actually have names for identification. These names can have a maximum of eight characters, like those of segments, and can have descriptors like `“.text”`, `“.data”` or `“.bss”`. Some of them are generated directly by the compiler and others are created by the linker. One of the segments contains, among other information, the application's entire code, all the data, all import and export information and all relocation information.

Structure of a PE file



Not all of the data fields within the many structures of a PE file are described in detail in the following discussions. A few have no clearly defined purpose. The variable names occasionally don't mean much. Furthermore, it's not always clear flags they mean when they talk about flags in one form or another. However, with the information that is described, it is possible to work along to the areas that are important for executing an application, for example the imported and exported functions.

Mapping to memory

We must talk about mapping PE files to memory before examining the PE format. The PE format was designed from the beginning so that PE files would only have to be mapped to memory so that the code they contain can be executed. To do so, we need the cooperation of the compiler when it is generating the code, as will become evident later in this chapter. The compiler must make certain the program code doesn't contain too many fixed addresses. The starting address of a PE file isn't always known when it is mapped to the process memory. In Windows 95, it is usually set by the linker at 0x400000 (4 Meg) for EXE files. However, you never know where DLLs will appear in memory.

Most of the pointers within the data structures of the PE format that point to other elements of the file shouldn't be considered as absolute memory addresses. Instead, they refer to the beginning of the file. Therefore, they're called a *Relative Virtual Address* or *RVA*. For example, 256 bytes away from the beginning of the file has the value 0x100 (256) in one of these pointers. If you want to calculate the address of the corresponding memory location when mapping the file to a particular address space, simply add the mapping's starting address to the RVA address. If the mapping starts at 0x400000, then the element for which the RVA pointer indicates 0x100 is at address 0x400100 in memory. So, the equation is the following:

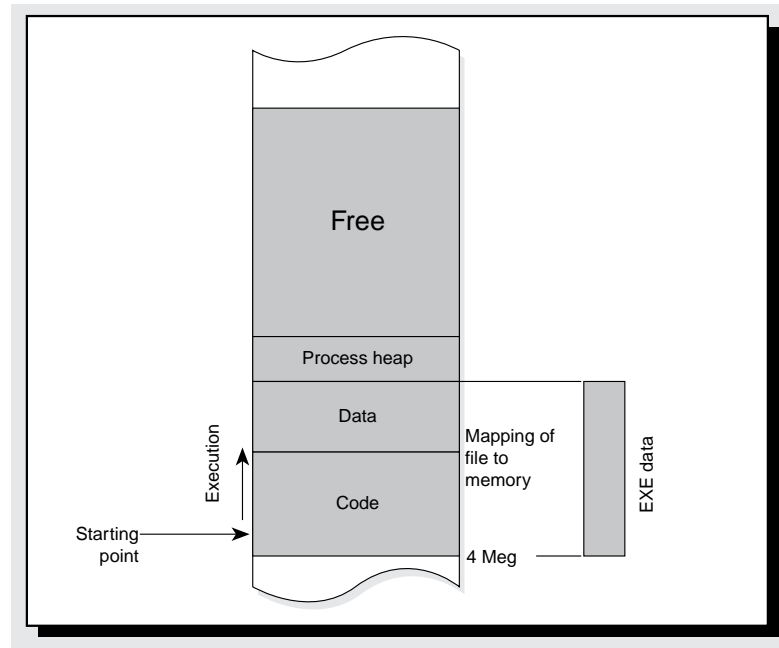
$$\text{Virtual address for memory access} = \text{RVA} + \text{base address}$$

The only question is, where do you get the base address of a process? The answer is quite simple. Take it from the copy handle of the process that you get when you start an application. The copy handle in Windows 95 consists of the base address of a process in memory. It's no longer an unambiguous value that you could use to distinguish between different copies of an application. However, this information is no longer needed because the system core always knows with which process it's working. After all, it executes the switching between the different processes. So, the above equation can be formulated even more concretely:

$$\text{Virtual address for memory access} = \text{RVA} + \text{hInstance}$$

In this connection, it's interesting to note that PE files are always completely mapped to memory. This includes not just the code and data sections, but the entire header and everything else in the file. Since a PE file always starts with a DOS header, you also see the DOS header first at the base address of a PE file in the process memory.

Mapping and executing a PE file in memory



What is impressive about this is that although all of the information from the PE file is still accessible during execution, no memory in RAM is used. That's because all sections that aren't accessed during execution of the process have to give way to other pages. In the course of the normal memory management, they are swapped out and aren't loaded back in because they are called for again. Furthermore, space isn't wasted in the swap file because the memory map is taken directly from the PE file itself. This is really an impressive trick.

The DOS header

The DOS header is what you'll see first in a PE file. It has the identifier "MZ" at the beginning. This header is needed so that a PE file can also be started in DOS. Along with the header, it also contains a small code. When this code is executed it outputs the message that the application can only run from Windows, and then it immediately terminates the application. Without this header, DOS would balk at starting the application because there is no MZ identifier and you wouldn't have any idea what the problem was.

//At the beginning of a PE file, there is a normal DOS header.

```
typedef struct {
    WORD  e_magic;                // Magic number "MZ"
    WORD  e_cblp;                // File size MOD 512
    WORD  e_cp;                  // File size DIV 512
    WORD  e_crlc;                // Number of segment addresses to be adapted
    WORD  e_cparhdr;             // Size of header in paragraphs
    WORD  e_minalloc;            // Minimum number of additional paragraphs required
    WORD  e_maxalloc;            // Maximum number of additional paragraphs required
    WORD  e_ss;                  // Contents of SS register at start of program
    WORD  e_sp;                  // Contents of SP register at start of program
}
```

```

WORD  e_csum;                // Checksum over the head of the EXE file
WORD  e_ip;                  // Starting offset for IP at start of program
WORD  e_cs;                  // Contents of CS register at start of program
WORD  e_lfarlc;              // Address of Relocation-Table within the file
WORD  e_ovno;                // Overlay number
WORD  e_res[4];              // reserved
WORD  e_oemid;               // OEM ID
WORD  e_oeminfo;             // reserved for OEM
WORD  e_res2[10];           // reserved
LONG  e_lfanew;              // Offset to subsequent PE header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

The DOS code for outputting the error message follows the DOS header. Since the actual PE header is next, you must know the size of the code. Because that information cannot be obtained directly from the DOS header, use the offset of the PE header relative to the start of the file instead. It's in the field `IMAGE_DOS_HEADER.e_lfanew`.

The PE header

You'll find a structure of the type `IMAGE_NT_HEADERS` here. Besides a signature identifying the PE file as a PE file, it contains two fields. These two fields themselves contain data structures: `FileHeader` and `OptionalHeader`. These structures are not optional but are always indicated. The signature must contain the ASCII string "PE\0\0" for Windows 95 can recognize the program loader. In other words, "PE\0\0" means "PE" in the low word and 0x0000 in the high word.

```

// PE header as the combination of two large data structures

typedef struct {
    DWORD Signature;                // always "PE\0\0"
    IMAGE_FILE_HEADER FileHeader;    // the actual header information
    IMAGE_OPTIONAL_HEADER OptionalHeader; // additional header information
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;

```

The field `FileHeader` contains a data structure of the type `IMAGE_FILE_HEADER`. It contains much important information, including the type of CPU for which the code contained in the file was compiled, the number of segments within the file and flags which identify the file type.

```

// The first part of the PE header

typedef struct {
    WORD  Machine;                // CPU, which can execute the program code
    // 0x0000 = unknown
    // 0x014C = 80386 and above
    // 0x014D = 80486 and above
    // 0x014E = Pentium and above
    // 0x0162 = MIPS R3000
    // 0x0166 = MIPS R4000
    // 0x0x16 = MIPS R10000
    // 0x0184 = DEC Alpha
    // 0x01F0 = PowerPC
    WORD  NumberOfSections;        // Number of sections in the file
    DWORD TimeDateStamp;           // Time of creation
    DWORD PointerToSymbolTable;    // Pointer to symbol table for debug info
    DWORD NumberOfSymbols;         // Number of symbols in the symbol table
    WORD  SizeOfOptionalHeader;    // sizeof(IMAGE_OPTIONAL_HEADER)
    WORD  Characteristics;         // Flags

```

```
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

The flags are set up as a bit field. Several flags in this bit field can be combined with one another. The most important of these is bit 13. Other than from the file name, how you can tell if it is an executable file or a DLL is through bit 13.

Bit	Meaning if set to 1
0	The relation information was removed from the file
1	File can be executed; it was linked by the linker without an error message
2	No debug information about the line numbers is available
3	No debug information about the program symbols is available
4..11	Additional flags, meaning unknown
12	This is a system file
13	The file is a DLL rather than a program
14..15	Reserved

The second data structure in the PE header is the `IMAGE_NT_HEADERS.OptionalHeader` field. It contains a data structure of the type `IMAGE_OPTIONAL_HEADER`. However, at least for EXE files and DLLs, this structure is extremely critical. It's packed full of important and interesting information. For example, this is where you determine:

- Where a file in memory is to be loaded, or in other words the recommended mapping start.
- Where the execution of the application should start; the starting address of the code.
- The earliest version of the operating system that is required to execute the application.
- How large the stack can grow and how much memory is committed just when the process starts.
- How much memory is committed for the process heap and how much is already committed when the process starts.
- Whether you're dealing with a console or a GUI application.

```
// The first part of the PE header
```

```
typedef struct {
    WORD    Machine;                // CPU, which can execute the program code
    // 0x0000 = unknown
    // 0x014C = 80386 and above
    // 0x014D = 80486 and above
    // 0x014E = Pentium and above
    // 0x0162 = MIPS R3000
    // 0x0166 = MIPS R4000
    // 0x0168 = MIPS R10000
    // 0x0184 = DEC Alpha
    // 0x01F0 = PowerPC
    WORD    NumberOfSections;       // Number of sections in the file
    DWORD   TimeDateStamp;          // Time of creation
    DWORD   PointerToSymbolTable;   // Pointer to symbol table for debug info
    DWORD   NumberOfSymbols;        // Number of symbols in the symbol table
    WORD    SizeOfOptionalHeader;   // sizeof(IMAGE_OPTIONAL_HEADER)
    WORD    Characteristics;        // Flags
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

The last field in the structure is `DataDirectory`. It represents an array with 16 elements of the type `IMAGE_DATA_DIRECTORY`. This is a small structure that indicates the RVA starting address of a segment within the PE file and its size. Of course, a PE file usually contains less than 16 segments, so the size 0 appears in individual array entries. The number of the segments is indicated by the field `NumberOfRvaAndSizes`.

```
// Format of the entries in the array IMAGE_OPTIONAL_HEADER.DataDirectory

typedef struct {
    DWORD   VirtualAddress;           // RVA starting address in the file
    DWORD   Size;                    // Size of section in bytes
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

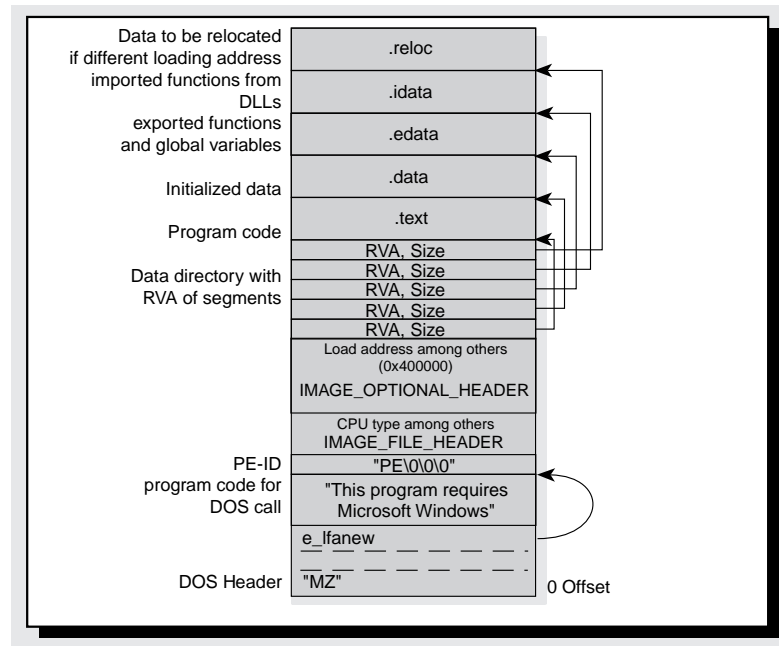
Segment table

The segment table follows the PE header. It describes the segments contained in the file. It corresponds with the entries in `IMAGE_OPTIONAL_HEADER.DataDirectory`. It always contains 16 entries. If there are fewer than 16 segments, only the first *n* entries are used; all remaining segments are filled with zeroes.

You'll first see the name of the segment inside the different entries. This name is always expanded to a size of eight letters using spaces. The different sizes and addresses are next. They refer to the position and size of the segment within the file one time and to its later position and size in memory another time. A definite difference exists between the two. The linker has seen the later virtual address space when the program is created. So, it has adjusted all relative address references to it.

However, in certain circumstances, the organization might appear slightly different in the file because segments don't have to be stored in their full later size. One example of this is the `.bss` segment. It's supposed to provide room in memory during execution of the program for all non-initialized global and static variables. So, the actual contents of this segment don't even have to be first stored in the file. After all, the variables that are to be generated all contain 0 at first. The actual segment descriptor is sufficient.

Organization of the PE segment table and associated C structures



Multiple views

Naturally this has tremendous implications for the loading process. It means the loader can't map the complete file to memory with just one view, i.e., one call to `MapViewOfFile()`. Instead, it first maps to memory the complete header, with the DOS header, the PE header and the segment table starting at the prescribed load address. Then it runs through the segment table and it uses the linker data for each segment to generate a memory map through `MapViewOfFile()`.

```
// For each section contained in the file, there is a structure
// of the type IMAGE_SECTION_HEADER.

typedef struct {
    BYTE    Name[8];           // Section name, filled with spaces for 8 characters
    DWORD   VirtualSize;       // Size of section in bytes
    DWORD   VirtualAddress;     // RVA starting address of section
    DWORD   SizeOfRawData;      // Size of file after rounding off to
                                // a multiple of 512 bytes
    DWORD   PointerToRawData;   // RVA where the section starts in the file
    DWORD   PointerToRelocations; // no meaning in EXE and DLL files
    DWORD   PointerToLinenumbers; // no meaning in EXE and DLL files
    WORD    NumberOfRelocations; // no meaning in EXE and DLL files
    WORD    NumberOfLinenumbers; // no meaning in EXE and DLL files
    DWORD   Characteristics;    // Contents of section
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

To generate the map using `MapViewOfFile()`, the loader requires the following:

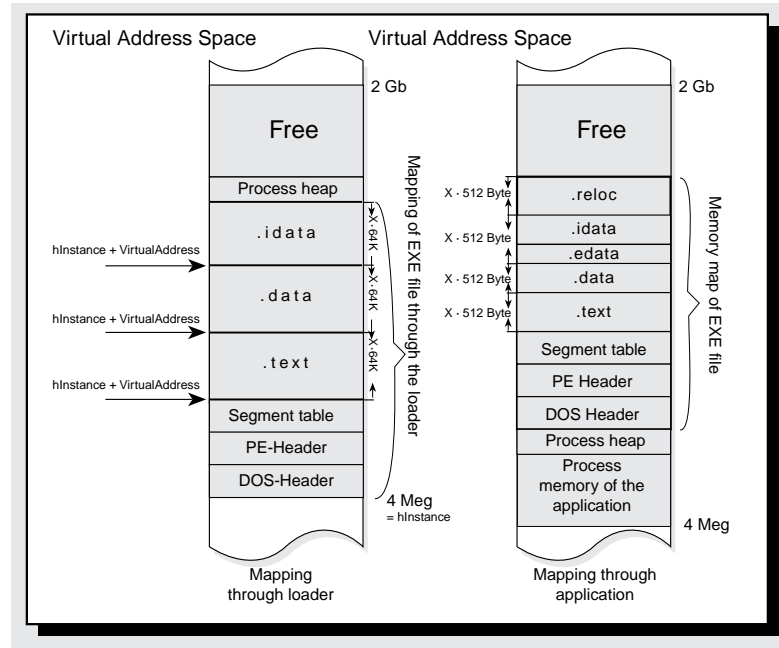
1. The later size of the region in memory and its starting address
2. The position of the segment within the PE file and its size there.

The following table shows the fields where the loader finds this information:

Feld	Meaning
VirtualSize	Specifies the size of the segment in memory, this figure is rounded off to a multiple of the allocation granularity of 64K during the concrete mapping.
VirtualAddress	The later starting address in memory as RVA. The section is mapped to the starting address <code>RVA + hInstance</code> .
SizeOfRawData	Size of segment contents in the PE file. It is already rounded off to sector size (512 bytes).
PointerToRawData	Offset of the segment contents in the PE file.
	Offset of the segment contents in the PE file.

By the way, the linker adjusts the entries in the segment tables so the individual segments are in the same order in the virtual memory as they are in the segment table. Although they immediately follow one another, they're always adjusted to the nearest 64K border. This means that the memory for them is allocated using `VirtualAlloc()` and the virtual memory management gives this granularity.

Difference between mapping a PE file through the program loader and a private mapping



That's why it makes a difference if you look at the mapping of a PE file the way the loader generated it in an application's address space or if you map a file to memory completely by itself. Then the DOS and PE header and the segment table will still be in the same location. However, the following segments themselves aren't in the same location. If you want to access a segment, you must get the necessary pointer using the `PointerToRawData` element and not through `VirtualAddress`. If you want to use pointers that are indicated within the different segments, you must convert them first. After all, they refer as RVA addresses to the later memory map of the file and not to the file offset and its one-to-one mapping in memory.

To get from an RVA address to the offset of the appropriate memory location within the file, calculate the difference between the start of the segment in virtual memory and its actual start within the file. Use the following equation:

$$\text{File offset} = \text{RVA pointer} - (\text{IMAGE_SECTION_HEADER.VirtualAddress} - \text{IMAGE_SECTION_HEADER.PointerToRawData})$$

To arrive at the real address of the memory locations regarding the current mapping of the file, add the file offset to the start of the mapping.

$$\text{Virtual address in memory} = \text{file offset} + \text{starting address of the mapping}$$

The program at the end of this section describes a program that reads out the PE file by direct mapping to memory. That's done much more easily than you might at first believe.

However, let's first return to `IMAGE_SECTION_HEADER`. Sometimes the most important information when you're looking for a segment is at the very end of the structure, in the `Characteristics` field. It tells about the contents of the segment, and it is needed so the program loader can recognize the code and data segments of a file. Contrary to what you might think, this organization isn't done based on the segment name, which each compiler can select as it wishes. Although the segments with the actual program code are always called ".text" with Microsoft, they could have a different name. The only way the loader recognizes them is by the `Characteristics` field. The following table shows the most important flags that are used in interpreting this field.

Constants	Value	Definition
IMAGE_SCN_TYPE_NO_PAD	0x00000008	Reserved.
IMAGE_SCN_CNT_CODE	0x00000020	Segment contains program code.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	Segment contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	Segment contains uninitialized data.
IMAGE_SCN_LNK_OTHER	0x00000100	Reserved.
IMAGE_SCN_LNK_INFO	0x00000200	Segment contains more information, i.e., for the debugger
IMAGE_SCN_LNK_REMOVE	0x00000800	The segment should not be mapped in memory.
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	Segment contains relocation data.
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	Do not lock the contents of the segment.
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	Do not cache the contents of the segment.
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	Do not swap out the contents of the segment.
IMAGE_SCN_MEM_SHARED	0x10000000	Segment cannot be shared by other processes.
IMAGE_SCN_MEM_EXECUTE	0x20000000	Segment is executable.
IMAGE_SCN_MEM_READ	0x40000000	Segment can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	Segment can be written to.

Segments in a PE file

If you look at executable Windows 95 applications with DUMPBIN, you will see repeatedly some segments with the same names and having the following contents:

Name of Segment	Contents	Name of Segment	Contents
.text	Program code of an application	.edata	Export definitions
.data	Initialized global and static data	.reloc	Addresses to be relocated
.bss	Uninitialized static and global variables	.debug	Debug information
.rdata	Description string from a DEF file	.tls	Thread local variable
.idata	Import definitions	.rsrc	Resources

*This is how
DUMPBIN sees
the segments of
a PE file*

```

C:\N>dumpbin wordpad.exe
Microsoft (R) COFF Binary File Dumper Version 3.00.5270
Copyright (C) Microsoft Corp 1992-1995. All rights reserved.

Dump of file wordpad.exe
File Type: EXECUTABLE IMAGE

Summary
 1000 .bss
 1000 .data
 3000 .idata
 7000 .rdata
 3000 .reloc
 A000 .rsrc
17000 .text

C:\N>

```

Some of these segments don't really need any explanation: .text contains the entire program code for the application .data contains all initialized global and static variables, like all string constants. The uninitialized global and static variables are

in the .bss segment. Since the data don't have any default values, you don't have to store them in the file. Therefore, the actual segment doesn't even exist. This is indicated with `SizeOfRawData` being equal to 0.

The structure of the other segments isn't officially documented, so it was not pursued further. This is true for the .debug segment with the debug information, the .tls segment, in which the thread local variables declared with `_declspec(thread)` are stored and the .rsrc segment with an application's resources. The .rdata segment contains only one thing: the name of the program or the DLL as it is given in the .DEF file using the `DESCRIPTION` command.

Relocated addresses

With the section on relocated addresses, we can answer the question of to what extent does the entire process not affect the compiler at all. Must the compiler use its code regeneration to let the code be mapped from out of the PE file to different processes without much effort? It must, because having different processes also means having different starting addresses for the program code and the data in the different process spaces. This is where the problem starts for the compiler. It becomes problematical for all data accessions within the program and all function calls that include direct address information. These are very basic commands that you'll find many times in any program.

```
PUSHD xxx [0x0004073C2] ; DWORD from memory location [0x0004073C2] to stack
CALL 0x00080FF24 ; Call procedure at address 0x00080FF24
MOV [0x000401234], EAX ; Place return value in memory location [0x000401234]
```

If the code in the address space shifts up or down by a few Meg, the absolute address citations in these commands must be adjusted. Their "target" is moving, just as they are. This adjustment process is called *relocation*. That's why the linker stores the addresses of all memory locations that have to relocate in the .reloc segment. This includes the addresses in machine language commands as well as pre-initialized pointers like those to string constants. The program loader adjusts the program code and parts of the initialized data appropriately during the loading process. This takes care of the entire matter.

However, this is not true for the virtual memory manager. This procedure represents much work for the VMM. Any of the pages in which changes were made can't be shared any more by different process instances. After all, now they are different for each instance. For example, where one instance has `MOV AX,0x00001234`, the other one says `MOV AX,0x00005678`.

Depending on how many pages are affected by the relocation, the efficiency of the entire PE idea can suffer greatly. So, when the compiler is generating code, it needs to try to create as little code as possible that needs to be relocated. Use the debugger to see how well that works. Several absolute addresses no longer present a problem. This is because the compiler simply indicates offsets now rather than absolute addresses for jumps and function calls (`JMP` and `CALL` commands). So it's using the distance to the jump target. Furthermore, this distance doesn't change with different starting addresses in memory. It's similar to an RVA.

On the other hand, you can't do anything about initialized pointers and access to global variables. However, you can use a little trick so the linker makes certain this is the exception (at least for EXE files). Have the linker relocate the addresses to the standard loading address for EXE files in Windows 95, 0x4000000 (4 Meg) when performing the links. If an EXE file is mapped to this address, it doesn't need to be relocated.

Imports

Use the .idata segment to determine which DLL functions a program or a DLL file is calling from other DLLs. This is where all imported DLL functions and the imported global variables are registered. However, after the program has loaded, you'll find this segment contains not only the names but also the actual addresses of the function in memory. That makes this segment extremely helpful for debugging and examining applications. A program at the end of this section uses its knowledge of how the .idata segment is structured to read the addresses of the DLL functions that are called from a running process.

DLL table

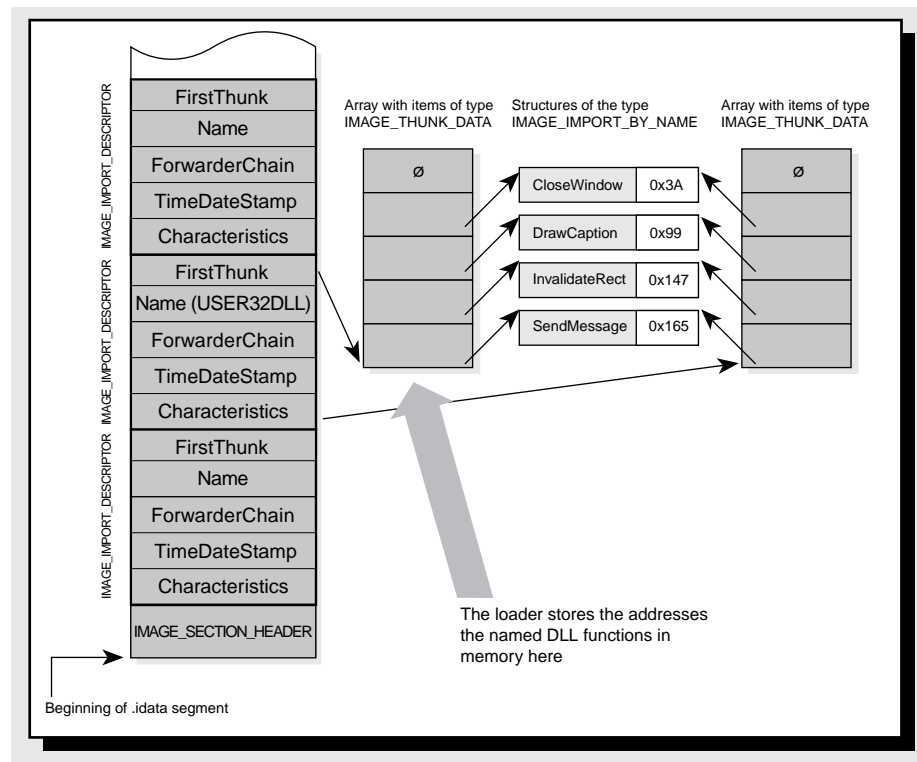
A DLL table appears at the beginning of the .idata segment. This is an array that contains an entry of the type `IMAGE_IMPORT_DESCRIPTOR` for each DLL that is to be imported. Although the exact number isn't stored, you can easily tell the end of the table because that entry consists completely of zeroes.

```
typedef struct {
    PIMAGE_THUNK_DATA Characteristics;    // RVA of 1st IMAGE_IMPORT_BY_NAME
                                          // array
    DWORD TimeDateStamp;                  // Time date stamp, usually 0
    DWORD ForwarderChain;                  // for forwarding calls
    DWORD Name;                           // RVA with address of DLL name as a C string
    PIMAGE_THUNK_DATA FirstThunk;         // RVA of 2nd IMAGE_IMPORT_BY_NAME array
} IMAGE_IMPORT_DESCRIPTOR;
```

The element Name contains an RVA that points to the DLL's name. The name is present as a C string at the indicated address in the mapping. Two arrays with entries of the type IMAGE_THUNK_DATA indicate which functions should be imported from a given DLL. The pointers Characteristics and FirstThunk represent the start of both arrays. Although these are separate arrays, their contents are identical at first. Each one contains exactly one entry of the type IMAGE_THUNK_DATA for each imported function. The program loader stores the addresses of the called DLL functions in the one array during the load process. These are the addresses the loader must determine so they can be called from within a given process.

The program loader first runs through the entries in the array to which Characteristics points. It gets the name of the desired DLL function from this array. Then the loader determines the address of the DLL function based on its mapping to the particular process memory. This is the address that it writes to the corresponding entry in the FirstThunk array.

Structure of the .idata segment and the associated C structures



So, only the real address of the function is stored there, rather than the DLL data. For example, if the third entry in the Characteristics array points to the `GetMessage()` function after the process starts, you'll find this function's address in memory in the third entry of the FirstThunk array.

It will soon become clear what is to happen with this address. First, let's look at the data structure `IMAGE_THUNK_DATA`. It represents the contents of both arrays.

```
// Elements of the arrays that Characteristics and FirstThunk point to

typedef struct {
    union {
        PBYTE ForwarderString; // unknown
        PDWORD Function; // real function address
        DWORD Ordinal; // Ordinal number
        PIMAGE_IMPORT_BY_NAME AddressOfData; // RVA of function name
    } ul;
} IMAGE_THUNK_DATA;
```

It merely consists of a union named `ul` in which four fields are mapped to the same memory location. Therefore, the program code must use the context to determine which field it wants to access. The meaning of `ForwarderString` is unknown. However, `Ordinal`, `AddressOfData`, and `Function` can be defined. `Ordinal` and `AddressOfData` tell the DLL function for which we're looking. The function with `Ordinal` is identified by its ordinal. The function with `AddressOfData` is identified by its name. Only one of the two pieces of information is used in the search. The highest bit in `Ordinal` (bit 31) determines which one is used. It must be set so we'll search for the DLL function with the ordinal from bits 0 through 30 of the field. If the bit isn't set, the name of the function we're looking for is determined through `AddressOfData`.

The element `AddressOfData` within the PE file represents the RVA of an `IMAGE_IMPORT_BY_NAME` structure. You can, from this, determine the name of the respective DLL function. However, during loading the loader overwrites the element `AddressOfData` in the `FirstThunk` array with the real function address. It can then be queried using the `Function` element.

```
// IMAGE_IMPORT_BY_NAME structures contain the name of a
// DLL file which is to be linked

typedef struct {
    WORD Hint; // potential ordinal number
    BYTE Name[1]; // Name as C string
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

The size of the `IMAGE_IMPORT_BY_NAME` structure varies because it depends on the name of the DLL function. The first byte contains the function's ordinal. However, consider this only as a starting point for the search and not as a definitive value. In contrast, the name of the function, which is a C string that follows the hint field, is definitive. Although the length of this string is shown to be 1, the structure accepts names of any length. So, the `Name` field only starts the string. Additional characters follow (including the concluding NUL character). You can do the following if you know the structure of the `.idata` segment:

1. Query all linked DLLs and the functions they call during loading
2. Query their addresses in memory while an application is executing

This makes everything very interesting regarding examining existing programs.

Exports

A listing of the exports in the `.edata` segment is much easier than a listing of the imports for at least two reasons:

1. You only have to cite your own exports.
2. You don't have to list the imports for various DLLs.

EXE files normally don't contain any exports because programs don't have anything to export. Programs are users and not servers. This is the complete opposite of DLLs, which never get by without an export segment. If a DLL doesn't contain any exports, you can't call up any of its functions.

The exports segment consists merely of an introductory header and three arrays that follow it. The arrays contain the name, the ordinal and the address of each function. It's normally a good idea with Microsoft products to pack something like this in a structure and build an array of these structures. However, they did things differently this time. If you find the name GetMessage() for which you were searching in the third entry of the name array, you'll find the associated memory address in the third entry of the address array and the associated ordinal in the third entry of the ordinal array. In other words, it's very simple.

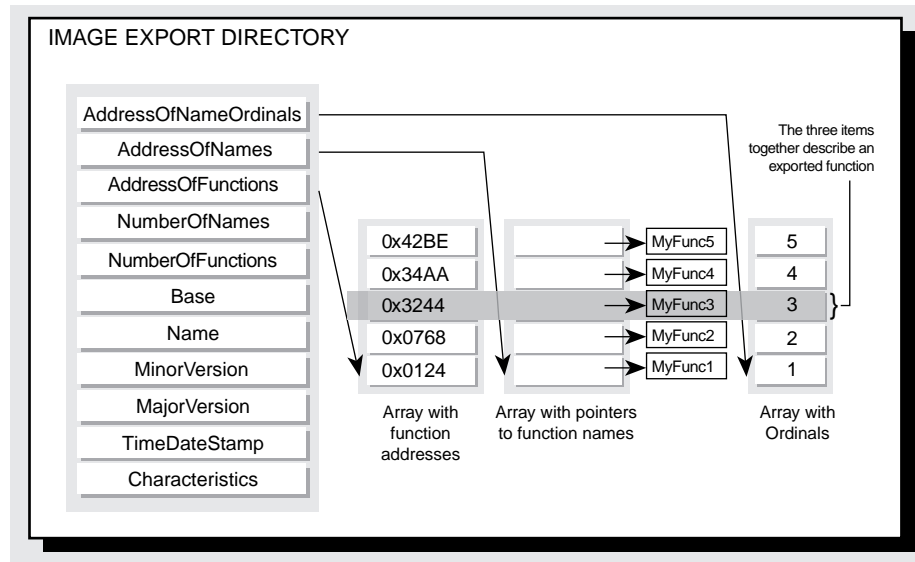
```
// IMAGE_IMPORT_BY_NAME structures contain the name of a
// DLL file which is to be linked

typedef struct {
    WORD    Hint;                                // potential ordinal number
    BYTE    Name[1];                             // Name as C string
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

You'll find the RVA of a C string with the name of the DLL in the Name field. This is, for example, the name the program loader looks for when running through the export segment. NumberOfFunctions contains the number of exported functions, which corresponds with the number of entries in the three arrays.

By the way, exported global variables at this level are treated like functions. So you don't see if you are dealing with a global variable instead of a function in one of the array entries. It will only become clear in the context of a later call as to with what you're dealing.

*Structure of the
.edata segment
and the
associated C
structures*



You'll find the three arrays at the addresses to which the three elements AddressOfFunctions, AddressOfNames and AddressOfNameOrdinals refer as RVA pointers. If you have run through these arrays, you know all exported functions of a DLL, their names and their addresses.

PE Monitor

We put together an example of how you can easily process files by memory mapping. The example is a small PE monitor appearing as a console application. The monitor displays two selected elements of a PE file: the import and export tables.

*An example of
PEIMEX.C
during execution*

```
00001820 6 BtmWndProc3d
00001900 9 ComboWndProc3d
00001940 15 Ct13dAutoSubclass
00001980 5 Ct13dColorChange
00001990 3 Ct13dCtlColor
00001C20 17 Ct13dCtlColorEx
00003520 19 Ct13dDlgFramePaint
00003C00 16 Ct13dDlgProc
00003E50 4 Ct13dEnabled
00003E80 0 Ct13dGetVer
00004120 22 Ct13dIsAutoSubclass
00001500 11 Ct13dRegister
00001550 18 Ct13dSetStyle
00000000 2 Ct13dSubclassCtl
00000000 24 Ct13dSubclassCtlEx
00001620 1 Ct13dSubclassDlg
00002440 20 Ct13dSubclassDlgEx
00001B90 23 Ct13dUnAutoSubclass
00001810 12 Ct13dUnregister
00001C50 25 Ct13dUnsubclassCtl
00001B10 21 Ct13dWinInitChange
00001F20 7 E414WndProc3d
00001730 8 ListWndProc3d
00001760 10 StaticWndProc3d
<key>
```

The program PEIMEX.C (PE-Import-Export) must evaluate the fairly complex structure of a PE file. Despite this, PEIMEX.C is unexpectedly simple. It consists merely of a main() function. First, the file is opened that was indicated on the command line after the program name when the program was called. Then an MMF object is created (CreateFileMapping()) using this file. The file is mapped in its entirety to memory using MapViewOfFile(). The starting address that is returned is stored in the pByte variable. Now you can use this pointer to easily make your way through the memory and, therefore, through the data.

Next, pByte is loaded to the pointer pDH, which is of the type PIMAGE_DOS_HEADER. It's thereby recognized as a pointer to a structure of the type IMAGE_DOS_HEADER. Use this pointer to retrieve the field e_lfanew from the DOS header and add the base address from pByte to it. Load the entire item in the pointer pNTH, which is of the type PIMAGE_NT_HEADERS. It points to a structure of the type IMAGE_NT_HEADER. Now you've worked your way to the PE header of the file.

Retrieve the start of the IMAGE_SECTION_HEADER and its number from this structure. Go through the IMAGE_SECTION_HEADER while searching for the .idata and .edata segments. This is all in memory and in the file.

If you have found one of the two segments, run through it the same way you did to work your way to the segment table. Interpret the different data structures and work your way from one structure to the next using the pointers contained in them. One important detail to remember is that all pointers within the data structures in the .idata and .edata segment of the PE file contain addresses relative to the memory mapping of the PE file the way the program loader did it.

In this case, however, we're working with the file that is on the hard drive. So, you need to calculate the difference between the actual location of the segment in the file and its location within an active process. This difference must be considered with following pointers. You can determine how great the discrepancy is from the different values in the IMAGE_SECTION_HEADER of a given segment. Then this difference simply must be subtracted from the pointers from the different data structures of a segment. That's how you get to the actual address of the particular structure within the mapping.

**You'll find the following program(s) on
the companion CD-ROM**



PEIMEX.C (C listing)



DDLs (Dynamic Link Libraries)

DLLs (Dynamic Link Libraries) are Windows' response to the disadvantages of *static linking* (also called *static binding*). The primary questions are how and when the machine code of a high-level language program will be linked with the functions called by the program from a runtime library. This occurs for static binding when the EXE file is created, when the linker combines the object code of the compiled program with the object code of the runtime library and writes it to a shared EXE file. In the process, an inseparable pair is developed. Afterwards, it's no longer easy to determine what the actual program code is and what originated from the runtime library.

Although static binding works well, it has disadvantages in specific constellations. For example, bound functions from the runtime library cannot be changed after static binding. If you detect an error or wish to change or add to the functionality of a function, you must create a new EXE file and distribute it to the users. Also, the same code of the runtime function is linked to many programs so if you start several programs simultaneously, the code will also be loaded several times into memory. That doesn't make much sense in a multitasking system. This only wastes valuable space in RAM.

Therefore, this is done differently with dynamic linking (dynamic binding). The program code doesn't become linked with the program code of the runtime library during linking. It's instead linked at runtime. The runtime library is provided for this purpose as a separate file, a Dynamic Link Library. At runtime, the program binds with the DLL so the functions contained within the library can be called from a preset mechanism.

When the runtime library undergoes changes, you don't have to recompile the complete program, since a replacement of the DLL file is sufficient. The next time the program is started, it automatically uses the changed functions. Also, the functions from the DLL usually aren't loaded because several programs can simultaneously reference one instance of the DLL in memory.

The application of this technology far surpasses the runtime libraries of the compiler. Any functionality that you do not want to make available to only one program is placed in a DLL. Because one program can bind with as many DLLs as it wants, divide the required DLL functionality among several DLLs with different areas of responsibility. This is how Windows works, providing its entire API system with more than a thousand functions available as several DLL functions. The following lists the best known examples:

- KERNEL32.DLL includes the system functions for memory management, for working with processes and threads, etc.
- USER32.DLL includes the routines for interacting with the user, for example, displaying windows, menus and dialog boxes.
- GDI32.DLL contains all the functions for drawing/painting and printing.

The following sections show you how to develop custom DLLs, use existing DLLs and discusses the intricacies of multitasking.

DLL Basics

Although Windows is familiar with only one kind of DLL, it understands two ways for binding a program to a DLL: During the loading process using the Shell or during runtime using calls to the corresponding API functions. We call the first example *load time dynamic linking*. The second example is called *runtime dynamic linking*.

Before looking at the details of these two ideas, the following summarizes DLLs' most important functions.

Win32 Functions for working with DLLs	
Win32 Functions	Task/Purpose
LoadLibrary	Loads a DLL or EXE in an application's address space
GetProcAddress	Returns a function's address within a DLL
FreeLibrary	Frees up a DLL previously loaded in an application's address space
FreeLibraryAndExitThread	Frees a DLL and exits the current thread
DLLEntryPoint	The entry point of a DLL, which is executed after loading
DisableThreadLibraryCalls	Suppresses the notification of a loaded DLL when attaching and detaching a thread
GetModuleFileName	Returns the filename of a loaded DLL or EXE
GetModuleHandle	Returns the module handle for a given filename
LoadLibraryEx	Extension of LoadLibrary()

Load Time Dynamic Linking versus Run Time Dynamic Linking

With “load time dynamic linking” the source code of a program already indicates that you want to use functions from a specific DLL by linking the appropriate Include file and simply calling the function. The DLL functions are handled within the program code exactly like external functions from other modules. Information about the required functions and DLLs is stored in the EXE file of the program. When the program is started, the Program-Loader automatically loads the required DLLs into memory and establishes the connection.

You need a LIB file with information about the functions contained in the DLL (more about that in the section on linking DLLs) so the compiler can take the appropriate precautions for the call of the DLL functions during compilation of the source code. Because the connection between application and DLL is static and won't be changed any more at runtime, we also call this *load time static linking*.

Load time linking is the preferred method. It's issued for calling most functions from the Win32 API because it's uncomplicated and as a programmer, you don't have to worry about anything. Use this method whenever you know exactly which DLL functions you want to call and in which DLL files the functions reside. On the other hand, runtime linking makes sense when the name of the required DLL library and perhaps even the names of the functions to be called in the library won't be known until program runtime. For example, Windows uses this method for communication with printer and multimedia drivers. Only from the Registry do the appropriate system services find out which driver is installed and, therefore, which DLL must be loaded for accessing the required functions.

Runtime linking requires the active cooperation of the application, because the functions to be called cannot already be declared within the source code. Instead, at runtime such an application first calls the `APILoadLibrary()` function to load a DLL file and bind to it. Then the application gets the addresses of the required functions within the DLL with the help of the `GetProcAddress()` API function.

```
HMODULE WINAPI LoadLibrary(
    LPCSTR lpLibFileName           // Name of DLL or EXE file to load
);
```

`LoadLibrary()` expects the name of the DLL or EXE file as its only parameter to be loaded in the address space of the current process. You'll usually specify the name of a DLL file here (see “DLLs as resource memory” for more information). If the specified filename doesn't have an extension, “DLL” is assumed to be the extension. If there is no search path specified either, various directories are searched for the file in the following sequence:

1. The directory from which the current process was loaded.
2. The current directory on the current drive.
3. The Windows system directory (usually `WINDOWS\SYSTEM`)

4. The Windows directory (usually WINDOWS)
5. The directories listed in the PATH environment variable

If the function could not find the desired DLL, it returns NULL. If it detects the DLL, the function checks first whether a DLL of the same name was loaded into the process from the directory. If so, the specified DLL does not need to be reloaded. Only the internal reference count of the currently loaded DLL is incremented. If no DLL of the same name was loaded, the function loads the DLL and maps it to the address space of the process. This also applies if a DLL of the same name was loaded but comes from a different directory. In this case, you get a module handle as a return value. You can address the DLL from now on using this handle. Module handles are of the HMODULE type, which corresponds to a LONG under the Win32 API.

Remember, these handles, unlike many other handles, are not inheritable. Thus, other processes cannot use this handle because the DLL to which it refers is not located in their address space. Then you can use the returned module handle to get the address of a desired function within the DLL. To do this, use the API `GetProcAddress()` function.

```
FARPROC WINAPI GetProcAddress(
    HMODULE hModule,                // Module handle
    LPCSTR lpProcName              // Name of function
);
```

Besides the module handle of the DLL, `GetProcAddress()` expects a pointer to a C string in the `lpProcName` parameter. The C string contains the name of the desired function. The notation and case must correspond exactly to what is in the DEF file of the DLL. DEF files are used in the development of DLLs to make the linker aware of the functions to be exported. See the section titled "From idea to DLL" for more information. Besides the name, however, you can also reference a DLL function using an ordinal number that the linker automatically assigns if you don't specify one explicitly within the DEF file. To search for a DLL function by ordinal number instead of by name, specify the number in the Lo word of the `lpProcName` parameter, while the Hi word has to be loaded with 0.

`GetProcAddress()` can distinguish both types of information because the ordinal numbers of the functions are basically smaller than the address of a C string. After all, the ordinal numbers range from only 1 to N (N = number of functions in the DLL), while the process memory of an application doesn't start until 4 Meg, so the pointer has to be greater than 4 Meg. Not even the Win32 API contains more than four million API functions. A maximum of 65536 functions per DLL can be exported, because the ordinal numbers are managed as 16 bit values. If the desired function was located, `GetProcAddress()` returns a pointer to it, otherwise it returns NULL. Use the API `GetLastError()` function to query for extended error information. With the help of the returned pointer you can then call the desired function easily. The following code excerpt will show this:

```
#include <windows.h>

void main()
{
    HINSTANCE hDLL;                // DLL Module handle
    BOOL (FAR PASCAL *lpGetClientRect)(HWND, LPRECT); // Pointer to FCT
    hDLL = LoadLibrary( "USER32.DLL" );
    if( hDLL )
    {
        // Get function pointer by name
        lpGetClientRect = GetProcAddress( hDLL, "GetClientRect" );
        // Get function pointer by ordinal number
        lpGetClientRect = GetProcAddress( hDLL,
                                           ( LPCSTR )MAKELONG( wOrdNumber, 0 ) );
        lpGetClientRect( hWnd, &r ); // Call function using pointer
    }
}
```

To remove a DLL loaded previously by `LoadLibrary()` from the process, use the `FreeLibrary()` function. This function decrements the reference count of the DLL and removes it from the process when the reference count reaches 0.

```
BOOL WINAPI FreeLibrary(
    HMODULE hLibModule           // Module handle of module to be unloaded
);
```

The only argument that `FreeLibrary()` expects is the module handle of the DLL, as returned during the preceding call to `LoadLibrary()`. If the specified module handle is valid, `FreeLibrary()` returns `TRUE` as the function result, otherwise it returns `FALSE`. Not only can `FreeLibrary()` be called by a process, it can also be called by a DLL that wants to unload itself. For example, a DLL might call `FreeLibrary()` as a reaction to a function call of the process, informing the DLL: “I don’t need you anymore”. If the DLL created several threads during its execution, which control its execution, using `FreeLibrary()` results in a small, logistic problem. Before the DLL calls `FreeLibrary()`, it needs to terminate all created threads first. However, without threads, the DLL cannot make a call to `FreeLibrary()`. So, at least one thread must remain to execute the call to `FreeLibrary()`. However, this not only removes the DLL from memory, it also extinguishes the thread. If this happens, the thread cannot exit clean. Still, there’s no other way, because if the thread exits first using `ExitThread()`, it won’t be able to call `FreeLibrary()` afterwards.

The `FreeLibraryAndExitThread()` function gives you a way out of this predicament, as it executes both tasks. First, the DLL is unloaded. Then the calling thread is terminated.

```
VOID WINAPI FreeLibraryAndExitThread(
    HMODULE hLibModule,           // Module handle
    DWORD dwExitCode             // Exit code of the thread
);
```

Therefore, as parameters the function expects both the module handle of the DLL to be unloaded and the Exit code for the current thread. The Exit code should be within the DLL. Otherwise, the remaining threads will be ended and, for example, won’t be able to free mutexes that are in their possession.

Initializing and terminating a DLL

Depending on a DLL’s internal structure, you may want to take note of the attachment of new processes or threads to the DLL, to provide resources or carry out initialization tasks as a precaution. The Win32 API gives you this option through what is called a DLL entry function. Although it’s not necessary, any DLL can define such a function. If the DLL makes use of this option, the system calls the DLL entry function every time a new process attaches to the DLL and detaches from it again. The same occurs when one of the attached processes generates a new thread or terminates a thread. The name of the DLL function is not predetermined by the system, but its syntax is:

```
BOOL WINAPI DllEntryPoint (
    HINSTANCE hinstDLL,           // Module handle of the DLL
    DWORD fdwReason,             // Reason for the call
    LPVOID lpReserved            // reserved
);
```

When the DLL file is being created from the individual object files, the linker must be informed whether a DLL entry function is to be provided, and what its name is. For this purpose, the linker has a special switch named `-entry:`, which is followed by the name of the DLL entry function.

```
link testdll.obj -out:testdll.dll testdll.exp - entry:MyDllStartfunction
```

Within the Visual C++ development environment, you specify a desired DLL Entry function by choosing Settings from the Project menu. In the dialog box that appears, select the “Link” tab. In the “Output” category you will find an input box called “Entry Point Symbol”. If this box is empty, the linker specifies the name of the DLL function as `DllMain()`. If you enter a

different name, the linker selects the specified function. If the DLL function is called, it gets the module handle as its first parameter, which identifies “its” DLL. This handle can be used in the following calls to one of the module functions from the Win32 API, for example, in a call to `GetModuleFileName()`. As the second piece of information, the DLL entry function finds out the reason for its call in the `fdwReason` parameter:

DLL_PROCESS_ATTACH

Displays the attachment of a process to the DLL at load time or during runtime as a result of a `LoadLibrary()` call. DLLs that are indirectly linked to the process by other DLLs also get this call. It occurs in the context of the application. The DLL can use this call as an occasion to perform initializations for its use under the process.

DLL_PROCESS_DETACH

Signals the detachment of the DLL from the process. The cause for this can be the termination of the process or an explicit call to `FreeLibrary()`. As a result, the DLL has the opportunity to free resources previously allocated for the process.

DLL_THREAD_ATTACH

Specified when the process generates a new thread. This doesn’t apply for the initial thread of a process for which, when it is being attached to a DLL, the DLL entry function was only called once with the `DLL_PROCESS_ATTACH` constant. So, before the first call with `DLL_THREAD_ATTACH`, at least one call has taken place with `DLL_PROCESS_ATTACH`.

DLL_THREAD_DETACH

Signals the termination of a thread whose creation was displayed earlier by `DLL_THREAD_ATTACH`. A call to the DLL entry function with `DLL_THREAD_DETACH` also occurs for the primary thread of an application for whose creation `DLL_PROCESS_ATTACH` was specified instead of `DLL_THREAD_ATTACH`. Then a call is executed with `DLL_PROCESS_DETACH`.

The third function parameter, `lpReserved`, is only important for the reception of `DLL_PROCESS_ATTACH` and `DLL_PROCESS_DETACH`. In this case, it indicates whether the DLL was loaded or removed dynamically or statically. In this context, dynamic means that the process explicitly called `LoadLibrary()` or `FreeLibrary()` to load and remove the DLL. If so, `lpReserved` contains `NULL`. However, if the DLL was statically loaded or unloaded, the value in `lpReserved` is not equal to `NULL`, i.e., the Shell places it in memory when the process is loaded and removes it from memory when the process is terminated.

```

/*****
/* DllMain: Main Entry Point of the DLL                                     */
/*-----*/
/* Parameter:      none                                                    */
/* Return value: TRUE - everything is okay                                  */
/*                FALSE - error occurred                                   */
*****/
BOOL WINAPI DllMain(HANDLE hModule, DWORD fdwReason, LPVOID lpReserved)
{
    switch (fdwReason)              /* Why was the function called?      */
    {
        case DLL_PROCESS_ATTACH:
            //
            // a new process has attached to the DLL
            //
            break;

        case DLL_THREAD_ATTACH:
            //
            // the current process has created a new thread
            //
            break;
    }
}

```

```

    case DLL_THREAD_DETACH:
        //
        // a thread of the current process has terminated
        //
        break;

    case DLL_PROCESS_DETACH:
        //
        // the current process is detaching from the DLL
        //
        break;
}
return TRUE;
}

```

By the way, you can disable the messages about the creation and termination of threads. This is a good idea if an application creates several threads during execution, then deletes them and wants to save a little time in the process. However, make certain that a DLL doesn't depend on knowledge of these processes. The appropriate function is called `DisableThreadLibraryCalls()` and expects as a parameter only the module handle of the DLL that no longer wants to be informed of the creation and termination of threads within its parent process. The function continues to receive information about attaching or terminating a process.

```

BOOL WINAPI DisableThreadLibraryCalls(
    HMODULE hLibModule           // Module handle of the addressed module
);

```

New and old API functions

You've already become familiar with the most important API functions such as `LoadLibrary()` and `GetProcAddress()`. Other functions include `LoadLibraryEx()` or `GetModuleHandle()`. Some other functions are worth discussing even though they no longer exist. `LoadLibraryEx()` is the intensified version of `LoadLibrary()`. Like `LoadLibrary()`, `LoadLibraryEx()` expects as its first parameter the filename of the DLL or EXE file to be loaded. However, there are two other parameters. Since the first one evidently plays no role, pass this parameter with `NULL`.

```

HMODULE WINAPI LoadLibraryEx(
    LPCSTR lpLibFileName,           // Name of DLL or EXE file to be loaded
    HANDLE hFile,                   // NULL
    DWORD dwFlags                   // decides about execution
);

```

The third parameter, `dwFlags`, provides leeway for some unconventional ways of working with DLLs. Under Windows 95, you can specify two constants:

LOAD_LIBRARY_AS_DATAFILE

Causes the specified DLL or EXE file to be mapped as is in memory, without taking any precautionary measures for executing the code contained within. This option is chiefly used to load resource DLLs into memory, which is a faster method than using `LoadLibrary()`. What is more, an attempt is not even made to call the DLL entry function.

LOAD_WITH_ALTERED_SEARCH_PATH

This flag lets you make slight changes to the order of directories to be searched for loading the DLL. While the search otherwise begins in the directory of the application, in this case the search starts in the directory from which the calling module originates. This makes sense, especially for DLLs which are loaded in a program and then want to load other DLLs. Sometimes

these other DLLs won't be located in the application's directory, but instead, are in the directory of the DLL that's starting the call, so specifying `LOAD_WITH_ALTERED_SEARCH_PATH` definitely makes sense.

`GetModuleFileName()` and `GetModuleHandle()` are two functions that provide users with information about a loaded module (DLL or EXE file). If you are interested in the filename of a loaded module, `GetModuleFileName()` will help. For a module specified by its module handle, the function loads the filename in a buffer provided by the caller.

```
DWORD WINAPI GetModuleFileName(
    HMODULE hModule,                // Handle of the module
    LPSTR lpFilename,              // Pointer to buffer for receiving the filename
    DWORD nSize                     // Size of the buffer in bytes
);
```

In the call to this function, a pointer to this buffer is expected as the second parameter. The third parameter is its size in bytes. The function recognizes whether the buffer has enough room for the filename based on the size specification. Because EXE files and DLLs under Windows 95 can have long filenames, make certain to provide 260 bytes for the buffer. If the buffer was too small or the specified module handle was invalid, you get a function result of 0, otherwise you get the length of the filename in the buffer.

`GetModuleHandle()` works in the opposite way. If you know or suspect that a specific DLL is loaded, but you don't know its module handle, use this function to determine the handle.

```
HMODULE WINAPI GetModuleHandle(
    LPCSTR lpModuleName             // Name of loaded DLL or EXE file
);
```

As an argument, the function expects a C string. The name of the module for which you are searching is placed in this string. Case is unimportant, and the function does not expect a path specification. If the function returns a value of `NULL`, the specified module is not loaded.

16-bit DLL functions

Several functions that were routine under the 16-bit API are no longer supported or only partly supported. This is because they don't fit into the new DLL idea of virtual memory management. For example, a look at the `WINBASE.H` file reveals that the following three functions no longer exist:

```
#define FreeModule(hLibModule) FreeLibrary((hLibModule))
#define MakeProcInstance(lpProc,hInstance) (lpProc)
#define FreeProcInstance(lpProc) (lpProc)
```

Fortunately, they won't cause us any problems during compilation because the following macros have been prepared for them: `FreeLibrary()` is automatically called now instead of `FreeModule()`. `MakeProcInstance()` and `FreeProcInstance()` only return the customary function result without resulting in a call to an API function.

The following three functions from the 16 bit API are no longer supported under Win32. This results in an error by the time it's linked because the linker cannot find them in the LIB files of the system.

```
GetCodeHandle()           // no longer exists
GetInstanceData()         // no longer exists
GetModuleUsage()          // no longer exists
```

Before the Win32 API was available, `GetCodeHandle()` was used to get the code segment in which a specific DLL function was located. Under the Win32 API this function is no longer necessary because all DLLs are in the one code segment that makes up the address space of a Win32 process.

The Win16 API used `GetInstanceData()` to copy the global data of a one DLL instance to another one. This is no longer possible, because the instances manage their global variables independently of each other, provided you don't create an additional data segment with shared global variables.

With `GetModuleUsage()` you used to be able to find out how many instances of a DLL or EXE file were in memory. You can't do this any more in Win32 API.

From idea to DLL

DLL files are produced by C modules in which the program code of the DLL functions to be exported is packed. As a result, the functions should be accessible to other programs. To this extent, DLLs are completely normal programs. The deciding difference is that they lack the inner drive to execution and their functions are called from the outside in reasonably loose order. Although it's possible to furnish DLLs with a start function, this function is not meant to keep the DLL running forever. Instead, this function is designed to initialize the important global variables, which serve as "cement" among the various functions within the DLL, just like in a normal C application. Immediately afterwards, it's best to go right back to the caller, because the longer the DLL spends in this routine, the longer the loading process takes.

The greater the functionality of a DLL, the greater the number of help functions that assist the actual DLL functions. Thus, a DLL won't want to export all functions, but only select ones. The corresponding functions have to be identified. The two methods available for this include one that is more traditional and one that is more modern. The more modern method is not an ANSI standard. In the traditional method, you create a "definition file", a simple ASCII file with the "DEF" extension.

After completing the C module, you create a text file, enter a few commands and save everything as a DEF file. As might be expected, it depends on which commands you enter whether the linker laughs at you or generates the desired DLL file. After all, it's the linker that merges the compiled C modules (OBJ files) and writes them together with the required export information to the DLL file. Therefore, you have to feed the linker the appropriate DEF file when it is linking the modules.

With a DEF file ...

DEF files must be written in a relatively plain syntax, which, however, provides a series of options as different commands. For example, a typical DEF file might look like the following:

```
LIBRARY "DllName.dll"
DESCRIPTION "my wonderful DLL"
EXPORTS
    Hello1
    Hello2
    Hello3
    Hello 4      <- no spaces please, that causes trouble
    Hello5 @5
    Hello6 @6 NONAME
```

Using the library command, first the name of the DLL file to be generated is specified. Then the `thedescription` command creates a string, which could, for example, contain a copyright or information about the version. The string is placed in the file and can be made visible with the help of the `DUMPBIN` tool. However, the important command is `exports`. You place the names of the functions to be exported after this command - one after the other, line by line.

In the case described above, 6 Hello functions are exported. The example uses two optional switches with `Hello5` and `Hello6`. They determine how the function is handled. An application has a chance through the `@` symbol to specify the ordinal number of the DLL function. This is the same ordinal number that you can specify with `GetProcAddress()`. Besides the name of an API function, it gives you the second option of referencing an external DLL function. However, you don't get very much out of this, and above all, the linker is quite strict with the ordinal numbers - at the very least the numbers must be single-valued. This the reason you normally do not specify ordinal numbers.

However, if your mind is set on working with ordinal numbers, you'll often consider concealing the real name of the function as well. It's no problem if you specify `nonname`. In this case, only the ordinal number of a function will be written to the DLL file. It's no longer possible to make inferences about the name of the function.

You can also export global variables by specifying their name together with the names of the functions after exports. Make certain to add data following the variable names so the linker doesn't believe it's dealing with a function. Also, other switches are available that can, for example, set the stack. However, we won't worry about these other switches for now.

```
LIBRARY "DllName.dll"
DESCRIPTION "my wonderful DLL"
EXPORTS

    ... Functions, functions and more functions

    g_Hello DATA
```

and without (DEF file)

If you're not against compiler specific directives and don't mind stretching ANSI compatibility, don't use the DEF file: `__declspec(dllexport)` is the name of the practical construct with which you can select functions and global variables for export directly to C code.

```
__declspec(dllexport) int Hello1( void ) {} // say Hello

/**** looks clearer this way *****/

#define DLLEXPORT __declspec(dllexport)
DLLEXPORT HMODULE Hello2( lpstr mes ) {} // you too
```

It's the same with global variables:

```
__declspec(dllexport) lpvoid lpvCopyBuf; // Pointer to copy buffer
```

Then you no longer need a DEF file. The compiler places the appropriate export information within the object file so the linker can access it. In this case the name of the DLL file must be specified in the call to the linker, like other settings that aren't supposed to match their defaults.

In the following DLL module, called `DLLMSGBOX.C`, two functions are exported using this method. The functions display a message. By specifying `__declspec(dllexport)`, `MessageBoxQuestion()` and `MessageBoxExclamation()` are automatically exported. You don't need a DEF file.

```
/*— DLLMSGBOX.C —*/

#include <windows.h>

__declspec(dllexport) int MessageBoxQuestion( LPSTR lpText )
{
    return MessageBox( NULL, lpText, "Question",
                      MB_ICONQUESTION | MB_YESNO );
}

__declspec(dllexport) void MessageBoxExclamation( LPSTR lpText )
{
    MessageBox( NULL, lpText, "Attention", MB_ICONEXCLAMATION | MB_OK );
}
```

Viewing the DLL file with DUMPBIN

You can quickly find out whether the desired imports appeared in the DLL file. Simply view the contents of the file using the DUMPBIN tool included with its C compilers from Microsoft. DUMPBIN, which replaced EXEHDR, reveals many interesting items about the contents of an EXE or DLL file. Read the chapter on the PE file format of Windows 95 for more information on this tool. Right now we're emphasizing the information about the exported data and functions. By calling

```
DUMPBIN mydll.dll /EXPORTS
```

DUMPBIN places the desired information within a console on the screen. Here's an example with a file called DLLMSGBOX.DLL.

```
Microsoft (R) COFF Binary File Dumper Version 2.55
Copyright (C) Microsoft Corp 1992-94. All rights reserved.
```

```
Dump of file dllmsgbox.dll
```

```
File Type: DLL
```

```
Section contains the following Exports for DLLMsgBox.dll
```

```

      0 characteristics
30444688 time date stamp Wed Aug 30 13:07:52 1995
      0.0 version
      1 base
      2 # functions
      2 # names
```

```

ordinal hint  name
      1      0  MessageBoxExclamation (00001000)
      2      1  MessageBoxQuestion (00001005)
```

```
Summary
```

```

1000 .bss
2000 .data
1000 .edata
1000 .idata
1000 .rdata
1000 .reloc
2000 .text
```

You'll find your functions and variables here if everything worked. The DLL is ready to be called by other programs. If you want to be amazed by the variety of functions in Windows 95, use DUMPBIN to look at the KERNEL32.DLL, SYSTEM32.DLL and GDI32.DLL system files. These are only three of the up to one hundred DLLs that you will find in some system directories.

DLLs well planned

The mechanism and idea of DLLs are one thing; the manner in which the developer converts them is another. Also, you must consider the questions of how to distribute the desired functionality to the functions of the DLL, name the functions and variables, design the header file, etc. This is especially true when a DLL is intended for third parties.

First, always combine logically coherent function groups in DLLs. This helps you maintain perspective when the jungle of functions, constants and types begins to grow. Also, this makes access easier for the user of the DLL. We've already seen that DLLs can contain hundreds of functions. We've only seen three examples (KERNEL32.DLL, GDI32.DLL and USER32.DLL). After all, such DLLs don't necessarily have to come from a single C source file, they can also be made up of dozens of different modules. It longer no matters once you have created the DLL.

It's also important the name be unique. When you put the file into a huge DLL directory like \WINDOWS or \WINDOWS\SYSTEM, the name shouldn't conflict with other existing DLLs. Don't be afraid to put some extra effort into creating the include file. You make this file available both to other developers for linking the DLL and to the call of the DLL functions. It's best if you use this file yourself for developing the DLL file so you must define all the types, constants and prototypes required for compiling the files in the include file. If you don't want to pass on all these elements to the caller later, simply put the definitions in a separate section of the include file. You can delete this section from the copy of the include file that you distribute to others.

Use extensive comments to clarify the meaning or significance of the various elements. This will be helpful to you in maintaining the library. It's also helpful for the users who will need the right information anyway if they're to understand the identifiers in the file. Be consistent in naming the functions, types and constants. Apart from the fact that it is advisable to use Hungarian notation, you characterize exported DLL functions through a constant prefix that refers to the area of responsibility and sometimes the name of a DLL. For example, use net for network functions or mail for mail routines.

Linking DLLs

Creating a DLL file is one thing but calling the functions contained in the file is another. To do this, you need the following:

DLL file Include file of the DLL file LIB file

You may not need the Include file if you know the function names or ordinal numbers. LIB files contain export information that the linker requires to analyze or break down the calls to the DLL functions within the program code. A LIB file is mandatory; there's no program and no access to the DLL files without a LIB file. However, this only applies for load time dynamic linking, that is, for the direct DLL calls in program code that the linker or loader has to break down or analyze. It's a different story with runtime dynamic linking, i.e., with calling LoadLibrary() and GetProcAddress(). These don't depend on the existence of a LIB file. They instead bind directly to the desired functions using the DLL file.

The LIB files for popular DLLs like KERNEL32.DLL are included with C compilers and are placed in their LIB directory. For example, the default directory for Visual C++ is \MSVC20\LIB“. The following program, called EXEMSGBOX.C, is an example. It calls the two functions MessageBoxQuestion() and MessageBoxExclamation() from the DLLMSGBOX.DLL file (from the previous section).

```
/*— EXEMSGBOX.C —*/

#include <windows.h>

// Prototypes of the functions exported by DLLMsgBox
int MessageBoxQuestion( LPSTR lpText );
void MessageBoxExclamation( LPSTR lpText );

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine,
                   int nCmdShow )
{
    if( MessageBoxQuestion( "Do you really want to see the second MessageBox?" )
        == IDYES )
        MessageBoxExclamation( "Okay! Here I am!");
    return 0;
}
```

You'll see the functions that the program imports by using DUMPBIN to look at the resulting EXE file (EXEMSGBOX.EXE). There are far more functions than the two from DLLMSGBOX.DLL. There's a series of system functions, which are called within the startup code of the runtime library. Along with the names of the functions, their ordinal numbers and the DLL file from which they were loaded are named. You can display the imports with the /imports switch, for example by calling

```
dumpbin exemsg.exe /imports
Microsoft (R) COFF Binary File Dumper Version 2.55
Copyright (C) Microsoft Corp 1992-94. All rights reserved.
```

```
Dump of file EXEMsgBox.EXE
```

```
File Type: EXECUTABLE IMAGE
```

```
Section contains the following Imports
```

```

DLLMsgBox.dll
    1  MessageBoxQuestion
    0  MessageBoxExclamation

KERNEL32.dll
    114  GetStartupInfoA
    D0  GetEnvironmentStrings
    EB  GetModuleHandleA
    9F  GetCommandLineA
    136  GetVersion
    62  ExitProcess
    1C6  RtlUnwind
    225  UnhandledExceptionFilter
    E9  GetModuleFileNameA
    92  GetACP
    F6  GetOEMCP
    98  GetCPInfo
    116  GetStdHandle
    DC  GetFileType
    232  VirtualFree
    230  VirtualAlloc
    24E  WriteFile
    E1  GetLastError

USER32.dll
    188  MessageBoxA
```

```
Summary
```

```

1000 .bss
1000 .data
1000 .idata
1000 .rdata
1000 .reloc
2000 .text
```

DLLs in the address space of a process

Regardless of whether a program uses load time or runtime dynamic linking to bind to a DLL, the loaded DLL is inserted into the address space of the process. As Chapter 40 showed us, the area from 2 GB (0x80000000) to 3 GB-1 (0xBFFFFFFF) in the address space of a process is reserved for the binding of DLLs. The DLL must appear in the address space of the process, because otherwise the code wouldn't be able to call the functions from the DLL. After all, a process in the flat-memory-model remains limited to its 32-bit address space. It's also blind to everything that is outside this area.

However, with the help of virtual memory management, the DLL is merely mapped to the address space of the process. Although it appears there, the DLL in reality is located in an entirely different area in memory. This information is concealed from the process. As far as the process is concerned, the DLL is where it's supposed to be. By mapping the DLL to the address space, the system gets the opportunity to make one DLL available in several processes simultaneously, without having to load the DLL several times. An appropriate setting of the Page tables for the processes is enough.

This mapping is cancelled once the process terminates or explicitly calls the API function `FreeLibrary()` to remove the DLL again. Calling one of the DLL functions after that would result in an error.

By inserting a DLL into the address space of a process, it becomes part of the process, a fact that is expressed in its relationship to the operating resources of the process. For example, the DLL functions use the calling thread's stack and can use all the handles created by the process (files, processes, GDI objects, etc.), provided the handles were passed to them as parameters of a function. The other way around, the process gets access to all exported global variables declared within the DLL. If the DLL allocates memory within one of the DLL functions, this memory is automatically created in the address space of the calling process. This guarantees the process can access this memory.

Managing global variables

Although the program code for all instances of the DLL in memory is identical, this is not true for the variables. Variables in this context means the global variables and the static variables within functions, the statics. Because each instance of the DLL in the different processes, together with its process and the other DLLs mapped within, represents a closed unit, it must include a separate set of global variables. As a rule, these variables are set as part of the response to the calls from DLL functions. The best example of this is a window handle, which the calling process passes to the DLL, so it can direct its output to the appropriate window for further calls.

Now several processes call the corresponding function, each with a different window handle - the handle of the particular process. If the corresponding global variable were shared by all the instances of the DLL, the window handles of the various processes would appear there. This is because one process after the other would call the corresponding function and leave its window handle behind. The last caller would be caught, because its window handle would remain in the global variable and the DLL would direct the output of other processes to its window. This wouldn't work because the system doesn't allow a DLL to access a window that belongs to another process. For this reason, each instance of a DLL automatically gets its own set of global variables. In switching between two processes, the global variables are automatically switched to the new DLL instance in the process. With the help of virtual memory management, this happens completely transparently, and one instance will never see the global variables of another instance.

However, it's possible to share some and sometimes all the global variables among all the processes. The loader doesn't create separate instances for these variables, so the different (code) instances of the DLL in the individual process can communicate with each other through these shared global variables. While the instantiated global variables go to the normal .data data segment, the shared global variables must be created in a separate segment, whose name can be selected. Within the source code, this new segment is introduced by a pragma (note: a pragma is a directive that instructs the compiler to implement features specified by the argument) named `data_seg`. The name of the segment must be specified (up to eight characters):

```
/* — DLLSOMETHING.C — */

#include <windows.h>

/* These global variables are placed in the .data-Segment and therefore */
```

```

/* are instantiated */
int g_instantiated;
long g_something;

/* the following global variables go in a separate segment named ".shar_gv"
#pragma data_seg( ".shar_gv" )
int g_shared;
long g_also_shared;
#pragma data_seg( ".data" )

/* the following data go back in the normal.data-Segment */

```

However, you cannot avoid DEF files if you want to use this option. Besides the usual information, you must insert a segment entry for the selected segment in the DEF file. The linker must determine what type of data is in this unfamiliar segment and whether the data is shared by various instances. Within the DEF file, you specify a section entry with the name of the segment and some attributes.

```

LIBRARY "DllName.dll"
DESCRIPTION "my wonderful DLL"
EXPORTS

... the various export functionsHello2

SECTIONS
    .SHAR_GV DATA READ WRITE SHARED

```

The `SHAR_GV` segment is declared as a shared segment (`SHARED` attribute), whose contents can be read (`READ` attribute) and written (`WRITE` attribute). Although you won't notice, the `.data` segment is assigned the read write attributes and `.text` is assigned the shared execute attributes. They contain the program code as well as the data to be instantiated.

DLLs as resource memory

Like other programs, DLLs can contain resources. These resources can be bitmaps, menus, dialog boxes or cursors. Sometimes they can even be used as resource memory, containing no code whatsoever next to the resources. For example, this is ideal for planning the localization of software products. You gather all the resources to be localized within one DLL. In other words, you don't pack them in the actual program. At program runtime you load the DLL with `LoadLibrary()` and use the returned instance handle in calls to `LoadString()`, `LoadBitmap()`, `LoadAccelerator()`, `LoadCursor()` etc. If you exchange the DLL for one with the same name containing the resources in a different language, the next time you start the application it presents itself in a different language. You may believe at this point that nothing could be easier. However, don't forget that it requires much patience and discipline for the developer to wait to load all the resources of the program at runtime.

Although you want to create a pure resource DLL, you will still need a C module. This module can be empty or consist of comments. Otherwise, the Make manager in Visual C++ won't be able to handle it. The resources are combined in an RC file, added to the project, as a result automatically compiled by the resource compiler and inserted in the DLL by the linker.

Multithreaded DLLs

DLLs by nature can serve several processes simultaneously. Therefore, it's also not difficult for them to work for several threads. After all, each process represents at least one thread. Expanding to several threads per process is the same as including more processes in the user list of the DLL. Nevertheless, it makes a difference whether only one thread or several threads per process use the DLL at the same time. The difference appears in the global variables of the DLL instead of the code of the DLL functions.

Unless otherwise specified in the DEF file, a DLL receives a separate set of global variables for each process into which it is inserted. When the DLL is loaded, the loader and the virtual memory manager are already taking care of this. If you switch to a different process, the current contents of the global DLL variables also change. So, the same DLL code sees different contents of the global variables in different processes. This ensures the different users of the DLL (the processes) don't conflict with each other.

Depending on the structure of a DLL, however, not only each process, but also each thread within a process requires its instance of these global variables. The Turtle DLL is a good example. If this DLL is called from several threads of a process, for each of these threads it maintains and draws in a separate window. To do this, the DLL requires the window handle for the output window, the current position of the turtle, the drawing color, etc. As long as the DLL only has to serve one process with an output window, it's easy to place all this information within global variables. Different draw functions simply take the starting point for the drawing operation from the global variable and place the end point there afterwards. When the next draw function is called, the DLL already knows where it is to continue drawing. If the process changes, the contents of the global variables also change. This prevents the different processes from conflicting regarding the next drawing point.

The same must also be true for several threads, at least if the turtle DLL is to be put in a position of serving several threads of a process simultaneously. You call this a "multi threaded dll".

You have several ways of avoiding the problem of global variables when called by multiple threads. The easiest way is simply not to use global variables. However, that means the different functions within the DLL cannot communicate through shared information or you would have to transfer management of these global variables to the callers of the DLL functions. In the case of the turtle DLL, you could set up an `InitThread()` function that each thread would have to call before it could use other functions of the DLL. Within this function you could allocate a memory block and store the necessary information in this block. This could include the current turtle position, etc. In other words, we're talking about all the information you would have managed as a global variable.

The function would return to its caller a pointer to this data packet and obligate the caller to specify the pointer for all calls of other turtle functions. The functions would gain access through this pointer to the global variables of their caller. Then they could separate the various threads. Ultimately, this is the same principle used by the Windows API when it requires the caller to pass a handle in many functions.

However, this can also be done differently. The Win32 API is familiar with an idea called Thread Local Storage (TLS). This idea creates global variables on a thread basis. However, this doesn't occur completely without the DLL being involved. Its functions must use various API functions for accessing and setting up TLS variables. This totals four API functions.

Thread Local Storage (TLS) idea

The TLS idea is based on an index memory that contains 64 DWORDs for each process that uses TLS memory. Within this memory, the process or DLLs loaded in the process can occupy one or more "slots" and store information there. However, the information per slot can amount to a maximum of only 4 bytes since the slots are managed as DWORDs. The contents of the slot change when you switch from one thread to another. This is because the TLS manager automatically manages the slot separately for each thread. Just as the global variables of a DLL automatically change when you switch between different processes, the different contents of the TLS slots also change every time you switch threads.

As a result, a DLL function only needs to know that it will find a required piece of information in slot x and can be sure to find a separate value there for each thread that it calls. You use TLS functions to access and allocate slots. The table on the right lists its functions.

A DLL uses `TlsAlloc()` to request TLS slots, one for each DWORD, which it must keep separate for each thread. If byte or word information is to be stored, several pieces of such information can be packed into one slot. However, then the code is responsible for preventing the different values from conflicting or an element from being overwritten during setting and reading of the slot. However, because there are enough slots available and

Win32 functions for working with TLS	
Function	Task
<code>TlsAlloc</code>	Make room for a thread local DWORD
<code>TlsSetValue</code>	Set value of a TLS-DWORD
<code>TlsGetValue</code>	Query value of a TLS-DWORD
<code>TlsFree</code>	Free thread local DWORD

in programming under the Win32 API you come into contact almost exclusively with `DWORD`s, you will generally want to save yourself the trouble of doing this. Call `TlsAlloc()` once for each `DWORD` to be stored, taking note of the return value. This value represents the slot number, in which the desired information is placed. You cannot specify this number when coding the corresponding C functions, but instead, have to use `TlsAlloc()` at program runtime to determine the number.

```
DWORD WINAPI TlsAlloc(
    VOID
);
```

To be on the safe side, always compare the slot number with the `TLS_OUT_OF_INDEXES` constant, which `TlsAlloc()` uses to report an error. This error message occurs when all available slots are already occupied, but this is hardly to be expected. From the moment of a successful `TlsAlloc()` call, the TLS mechanism also switches the contents of the slot with every thread change. Use the `TlsSetValue()` function to change the contents of the slot. This function expects the slot number as its first argument and an `LPVOID` pointer as its second argument. After the appropriate casting (conversion of data types), however, you could also place another data type here, which breaks down into a `DWORD` at compiler level.

```
BOOL WINAPI TlsSetValue(
    DWORD dwTlsIndex,
    LPVOID lpTlsValue
);
```

If the function returns `TRUE`, you can set the desired value of the slot for the current thread. This doesn't change the contents of that slot for other threads. If another DLL function wants to read out the contents of the slot, it uses the `TlsGetValue()` function. This function expects the slot number as an argument and returns the `DWORD` stored in this number. Again, this is only about the current thread.

```
LPVOID WINAPI TlsGetValue(
    DWORD dwTlsIndex
);
```

To free a slot allocated by `TlsAlloc()`, call `TlsFree()` and pass the slot number to the function. If the function was able to free the slot (return value of `TRUE`), it is no longer available for calls by `TlsGetValue()` and `TlsSetValue()`.

```
BOOL WINAPI TlsFree(
    DWORD dwTlsIndex
);
```

TLS in the TURTLE.DLL

Because of their different character, the four TLS functions must be used at different places within the DLL. This becomes clear if you look at the multi threaded version of the turtle DLL called `TURTLE.C`. The allocation of the TLS slots occurs within the `DllMain()` function, which is called as the start function of the DLL. Allocation of the slots occurs when this function receives the `DLL_PROCESS_ATTACH` constant as the reason for its call (`fdwReason` parameter). This is the signal that a new process has attached to the DLL. It's taken as the starting point for the allocation of the slots that are to become available for the different threads of the process. You won't know how many threads it will be at first, however, it's unimportant for setting up TLS memory. If `CreateThread()` creates new threads over the course of the program, they get their own copy of the various slots.

```
/* ***** */
/* DllMain: Main entry point of the DLL */
/* _____ */
/* Parameter:    none */
/* Return value: TRUE  - Able to pop stack memory */
/*              FALSE - Stack underflow */
```

```

/*****
BOOL WINAPI DllMain(HANDLE hModule, DWORD fdwReason, LPVOID lpReserved)
{
    switch (fdwReason)                /* Why was DllMain called? */
    {
        case DLL_PROCESS_ATTACH:
            /* A newer process wants to claim the services of the DLL. */
            /* Therefore, the process must allocate the required TLS indexes */
            /* for all threads running under its charge. */

            if( !(tlsidxX          = TlsAlloc()) ||
                !(tlsidxY          = TlsAlloc()) ||
                !(tlsidxAngle      = TlsAlloc()) ||
                !(tlsidxLineWidth  = TlsAlloc()) ||
                !(tlsidxColor      = TlsAlloc()) ||
                !(tlsidxPen        = TlsAlloc()) ||
                !(tlsidxWnd        = TlsAlloc()) ||
                !(tlsidxStackPtr   = TlsAlloc()) ||
                !(tlsidxStackMem   = TlsAlloc()) ||
                !(tlsidxBoundingLeft = TlsAlloc()) ||
                !(tlsidxBoundingTop = TlsAlloc()) ||
                !(tlsidxBoundingRight = TlsAlloc()) ||
                !(tlsidxBoundingBottom = TlsAlloc()) ||
                !(tlsidxUseBounding = TlsAlloc()) )
                return FALSE;
            return TRUE;
        break;

        case DLL_THREAD_ATTACH:
            /* The turtle must be initialized for each new thread —*/
            turtleInit();
        break;

        case DLL_THREAD_DETACH:
            /* After thread termination, turtle variables (stack, GDI objects)*/
            /* must be released. */
            turtleExit();
        break;

        case DLL_PROCESS_DETACH:
            /* When the process "dies", the TLS slots are no longer required */
            /* either. */
            TlsFree( tlsidxX );
            TlsFree( tlsidxY );
            TlsFree( tlsidxAngle );
            TlsFree( tlsidxLineWidth );
            TlsFree( tlsidxColor );
            TlsFree( tlsidxPen );
            TlsFree( tlsidxWnd );
            TlsFree( tlsidxStackMem );
            TlsFree( tlsidxStackPtr );
            TlsFree( tlsidxBoundingLeft );
            TlsFree( tlsidxBoundingTop );
    }
}

```

```

        TlsFree( tlsidxBoundingRight );
        TlsFree( tlsidxBoundingBottom );
        TlsFree( tlsidxUseBounding );
    break;
}
return FALSE;
}

```

The function calls `TlsAlloc()` more than 12 times, each time creating the slot for one of the variables that are to be managed thread-locally. The slot numbers are stored in variables such as `tlsidxX`, `tlsidxY`, `tlsAngle`, etc. They represent one of the required information items such as the current coordinate of the turtle or its direction of rotation. These are global variables defined at the beginning of the C file. Although we're trying to avoid global variables, a shared usage of these variables here by different threads won't hurt. This is what we want because the individual DLL functions determine the slot number from which they can get specific information through the contents of these variables. The specific information may be the current direction of rotation, for example. Besides, this slot number is the same for all threads; only the contents are different.

```

/** Start of TURTLE.C */

#include <windows.h>
#include <math.h>
#include <assert.h>

#include "turtle.h"

DWORD tlsidxX;           /* current x-coordinate */
DWORD tlsidxY;           /* current y-coordinate */
DWORD tlsidxAngle;       /* current angle */
DWORD tlsidxLineWidth;   /* line width */
DWORD tlsidxColor;       /* line color */
DWORD tlsidxPen;         /* current pen */
DWORD tlsidxWnd;         /* current output window */
DWORD tlsidxStackMem;    /* stack memory */
DWORD tlsidxStackPtr;    /* stack pointer */
DWORD tlsidxBoundingLeft; /* Bounding rectangle, left border */
DWORD tlsidxBoundingTop;  /* Bounding rectangle, top border */
DWORD tlsidxBoundingRight; /* Bounding rectangle, right border */
DWORD tlsidxBoundingBottom; /* Bounding rectangle, bottom border */
DWORD tlsidxUseBounding;  /* Adapt output to bounding rectangle */

```

Once the different slots are allocated, the individual DLL functions can access a slot using `GetTlsValue()` and `SetTlsValue()`. For example, functions can find out the current x-coordinate of the turtle of the thread using `GetTlsValue(tlsidxX)` and set the current direction of rotation using `SetTlsValue(tlsidxAngle)`. To increase the readability of the program code, these calls were placed in macros, which are also defined at the beginning of `TURTLE.C`:

```

/*== Macros for simplifying TLS access =====*/
#define GetX() ((int)TlsGetValue(tlsidxX))
#define GetY() ((int)TlsGetValue(tlsidxY))
#define GetAngle() ((float)(LONG)TlsGetValue(tlsidxAngle))
#define GetLineWidth() ((int)TlsGetValue(tlsidxLineWidth))
#define GetColor() ((COLORREF)TlsGetValue(tlsidxColor))
#define GetPen() ((HPEN)TlsGetValue(tlsidxPen))
#define GetWnd() ((HWND)TlsGetValue(tlsidxWnd))
// etc.

```



```

#define SetX(v)          assert(TlsSetValue(tlsidxX, (LPVOID)(v)))
#define SetY(v)          assert(TlsSetValue(tlsidxY, (LPVOID)(v)))
#define SetAngle(v)      assert(TlsSetValue(tlsidxAngle, (LPVOID)(LONG)(v)))
#define SetLineWidth(v)  assert(TlsSetValue(tlsidxLineWidth, (LPVOID)(v)))
#define SetColor(v)      assert(TlsSetValue(tlsidxColor, (LPVOID)(v)))
#define SetPen(v)        assert(TlsSetValue(tlsidxPen, (LPVOID)(v)))
#define SetWnd(v)        assert(TlsSetValue(tlsidxWnd, (LPVOID)(v)))

// etc.

```

In the Set macros, the TlsSetValue() call is bounded by an Assert call, so the program aborts with a warning if setting the value fails. Turtle functions such as turtleSetPen() use these macros extensively to set information such as the current pen, color and line thickness only in relation to the thread that called the functions.

```

/*****
/* turtleSetPen: Creates new current pen.
/*-----*/
/* Parameter:      crCol      : Color of new pen
/*                  lLineWidth: Width of pen
/* Return value: none
*****/
void WINAPI turtleSetPen( COLORREF crCol, LONG lLineWidth )
{
    /* free any existing pen
    if( GetPen() ) DeleteObject( GetPen() );

    /* Place pen in TLS
    SetPen( CreatePen( PS_SOLID, lLineWidth, crCol ) );
    SetColor( crCol ); /* Put color and width in TLS
    SetLineWidth( lLineWidth );
}

```

The slots are released in the same spot where they were created. (In our example this is in the DLL entry function, DllMain().) When this function receives the DLL_PROCESS_DETACH constant in fdwReason and finds out about its separation from the current process, TlsFree() releases all previously allocated slots. The slot numbers are obtained from the various global variables. In generating a new thread and the accompanying call of the DLL entry function with DLL_THREAD_ATTACH, the function just has to initialize the contents of the slots. This occurs with a call to turtleInit().

The TURTLE.DLL is used in a program called MULTIPLE. It's the most complex variation of the Pythagoras theme from the chapter on multitasking. A VB program called MULTIPLE, with the help of MULTIPLE.DLL, which was written in C, creates an optional number of windows. A Pythagoras tree is drawn in each of these windows. To draw the trees, MULTIPLE.DLL uses the functions from TURTLE.DLL. Intensive use is made of TURTLE.DLL's ability to serve several threads of a process simultaneously, because MULTIPLE.DLL creates two threads for each Pythagoras window to be displayed. One thread controls the drawing. A second thread is responsible for window management in the context of message processing.

The MULTIPLE.DLL shows that a DLL can be called by several threads of a process at the same time, but also is capable of creating threads on its own.

In this way you overcome the stigma of all DLLs, namely that their execution is only controlled from outside and they cannot become active themselves. For example, if you create a thread within the DLL entry function, it will be executed with the other threads of the

You'll find the following program(s) on the companion CD-ROM



MULTIPLE.C (C listing)
 MULTIPLE.DEF (C listing)
 TURTLE.C (C listing)
 TURTLE.H (C listing)
 TURTLE.DEF (C listing)

process equally. So, apart from all the DLL calls by the threads of the process, the DLL can remain active, for example to perform cleanup work in the background or wait for specific system events.

TLS memory - Microsoft style

MULTIPLE files		
MULTIPLE.EXE	MULTIPLE.DLL	TURTLE.DLL
MULTIPLE.VBP	MULTIPLE.C	TURTLE.C
MULTIPLE.FRM	MULTIPLE.DEF	TURTLE.H
	TURTLE.LIB	TURTLE.DEF

As normal components of the Win32 API, the four TLS functions can be used with all compilers and languages that allow calls to DLL functions. However, with its own language expansion, Microsoft is showing how much easier it is to integrate TLS variables into code without having to resort to the different TLS functions. This language expansion only works with the VC++ compiler (since version 2.0), for which it was exclusively implemented. The new identifier, `thread`, which must always be specified in conjunction with the `__declspec` directive, has the center stage. `__declspec()` signals the compiler that a Microsoft language expansion follows which doesn't meet the ANSI standard. In conjunction with the `__declspec(thread)` construct, you can declare global variables within a program as well as static variables within functions as thread specific:

```
__declspec(thread) int g_idX;                // okay, it's thread local
__declspec(thread) int __declspec(thread) g_idX;
                                           // doesn't work as Type Modifier

void func()
{
    __declspec(thread) static long cBits;    // works as a static variable
    __declspec(thread) char chKey;          // not as automatic local
}
/*— with a macro it looks a lot easier —*/

#define THREADLOCAL __declspec(thread)

THREADLOCAL int g_idX;                      // thread local

void func()
{
    THREADLOCAL static long cBits;          // thread local
}
```

`__declspec(thread)` cannot be added to the local variables of a function, however, these variables are thread local anyway, because every thread has its own stack on which these variables are placed.

The only way to avoid trouble with TLS memory is to place `__declspec(thread)` at the beginning. You no longer have to allocate any slots through `TlsAlloc()`, call `TlsGetValue()` or `TlsSetValue()` for each access to the thread local instance of the global variable `g_idX` and free the slot at the end through `TlsFree()`. The compiler does this automatically by providing the appropriate code. Access to the variable within the program code occurs like every other variable. However, the compiler adds invisible code that guarantees you're always operating with the specific instance of the current thread.

A small exception, however, does exist. If you use the `__declspec(thread)` construct in DLLs, they must be loaded by load time linking. This means that an application resorting to these DLLs cannot load them at runtime using `LoadLibrary()` and then procure the addresses of the desired DLL functions for the DLLs using `GetProcAddress()`. The internal code created by the compiler must be able to set the required slots for the individual DLLs of the application already when the process is loaded. This is also why this method cannot be used for DLLs that are directly called from VB or whose call occurs indirectly through a DLL, called in VB. VB always loads DLLs at runtime using `LoadLibrary()`. This is also why we had to resort to the conventional methods of TLS management in the multi-Pythagoras example.

If you use the `__declspec(thread)` construct in your DLLs, please warn users in the appropriate header files that the DLL cannot be loaded at runtime using `LoadLibrary()`. If you can't avoid loading the DLL at runtime, you must resort to the traditional method of TLS access using the four TLS functions.



Common Controls

The software industry is similar to other industries in that attention is closely paid not only to functional and technological improvements, but to satisfying the public's appetite for new products. Windows 95 is a perfect example, from the viewpoints of both the user and the developer. The trees, lists and icons appearing on your Desktop are intended not just for system applications but also for the client area. Therefore, you can use them in your programs. Microsoft seems to be promising the popular "Windows 95 compatible" logo only to programs that fully support the new elements of the Windows 95 user interface. Perhaps you don't need them to arouse your enthusiasm for the new look and feel of Windows 95.

This chapter concerns the new screen controls in the Windows 95 user interface. *Common controls* is used for these reasons:

1. *Controls* is used because they're managed as windows and controlled using messages like the familiar edit controls, buttons, and listboxes.
2. *Common* is used because they are available to all applications and not just the system.

This chapter talks about the three largest and most complex of the twelve common controls (TreeView control, ListView control and Image List). The Explorer uses these controls as the directory tree whose entries you can expand or compress, and the redesigned listbox that can change display modes. Complexity is inherent in the common controls. Many messages and supporting data structures are defined for communication between the controls and your program. Although most have usually have 10 or 20, the ListView control described here has almost 100. You have to be a little careful not to get so bogged down with its details that you lose sight of the overall functionality. Despite the many messages, you'll eventually become familiar with the ListView and its power and flexibility.

Once you start working with the controls, you'll find that learning a new one becomes easier. Unlike several other software interfaces from Microsoft, this one observes a fairly consistent naming convention for constants and structures. After working with the TV_ITEM data structure and TVM_SETITEM message of the TreeView control, you'll find the ListView's LV_ITEM data structure and LVM_SETITEM message much easier.

Manipulating the controls and getting data into them at runtime is important. However, we'll talk about another increasingly important aspect of the user interface in this chapter: *Drag and Drop*. Capabilities for moving controls with the mouse must be as intuitive as possible for the user. This is still true if dragging is allowed only within the current control or even a dialog box. Consider carefully whether it really makes sense for the user to arbitrarily position controls on the screen. Don't overuse this feature; keep in mind, however, the common controls in all cases provide the basic requirements for its use.

Accessing the common controls in the program code

The key to accessing the common controls in program code is in the COMMCTRL.H file (usually in the \MSTOOLS\INCLUDE directory). It contains all the declarations for working with the controls, that is, for the many types, messages, API functions and macros. This file must be included in the C module that will access the controls. In addition, the COMCTL32.LIB file is required, since it contains the interface definition for linking to the system file called COMCTL32.DLL. The latter file holds the code of the various API functions and window classes that represent the common controls.

The key concept is that of the window class. A common control is created at runtime by a call to the API function CreateWindowEx(). One of the predefined strings from COMMCTRL.H is specified for the window class, for example, "SysTreeView32" for a TreeView control. Subsequently, the common control is accessed just like a normal control. In other words, building a message with uMsg, wParam and lParam and then sending it to the control with SendMessage(). However, be careful: You must remember one call before CreateWindowEx(). This is the call to the InitCommonControls() function from COMCTL32.DLL. This initializes the common control window classes. Without prior initialization, a call to CreateWindowEx() will fail. You'll see more about this in the following chapters.

Creating controls in the dialog editor

If you want to have common controls appear as part of a dialog box, create them at design time in the dialog editor instead of at runtime with `CreateWindowEx()`. Using `CreateWindowEx()` is not only more cumbersome, but has another disadvantage. Controls created with `CreateWindowEx()` use the Windows default character set and not that of the dialog. Only after sending a `WM_SETFONT` message to the control will it use the desired dialog font.

The new common controls are not at first available in the dialog editor (at least in our versions of VC++, compiler versions 2.0 and 2.2). No amount of searching for hidden option switches will make them available. The key to accessing these controls is in the Registry. Here under `HKEY_CURRENT_USER\Software\Microsoft\Visual C++ 2.0\Dialog Editor`, the key `ChicagoControls` and value 1 are required. When you restart VC++ after adding them, the common controls will be available in the dialog editor. Microsoft may have fixed the problem by now. However, if you don't see the controls in your version, try the switch.

*The key for the
Windows 95
controls*

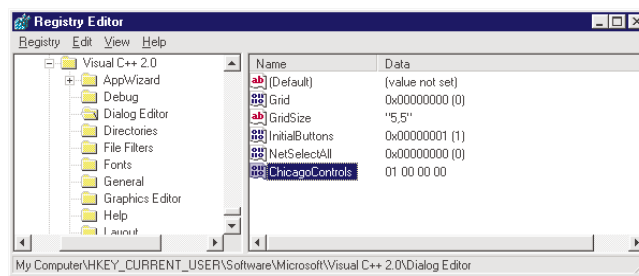


Image List

Your work becomes more difficult when a program must place more images on the screen in buttons or other controls. You'll need so many handles to manage the bitmaps, icons and cursors that you may not know where to put them all. Windows 95 addresses this problem with the new Image List control. Since it's a common control, it's available to all applications. Unlike other controls, an Image List is not visible. However, while it never appears on the screen, it can nonetheless be very helpful. Image Lists serve as holders for all types of images required in applications. These include small bitmaps placed on buttons, icons that symbolize files, etc. They can be easily referenced by a sequential number from an Image List and displayed on the screen.

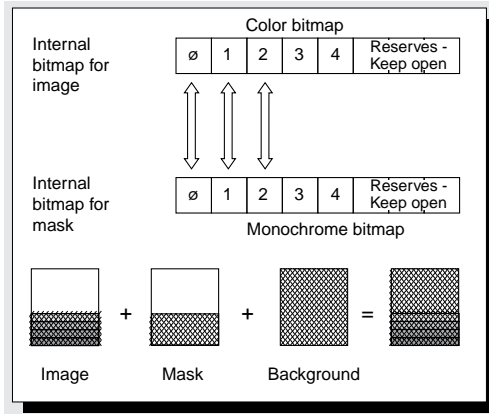
One prerequisite must be met for saving images in an Image List. All the images must have the same size. Otherwise, Image List won't work. This is the essential characteristic of an Image List. If you need to work with different size images, simply create a separate list for each format. Image Lists are used not only by programs but can serve various common controls as storage areas for their graphics. The TreeView and ListView, in particular, use these for displaying icons beside the text of the individual entries. You link the control to an Image List, then supply only the index position of the desired graphic for each node and not the graphic itself. This saves memory and much work.

Characteristics of an Image List

Unlike the other common controls, Image Lists are not controlled by messages but by a special API with about 30 functions. This is because they do not create windows as the other controls do. The functions are defined in the `COMMCTRL.H` file and can be called by a program that links in the `COMCTL32.LIB`.

The Image List's exception to the rule arises from the fact that it does not appear in windows and dialog boxes like other controls. Therefore, it cannot participate in the message processing within a window. Since you must find another way to talk to an Image List, use the DLL mechanism. You can recognize the various functions by the prefix `ImageList_` (for example, `ImageList_Add()` and `ImageList_Create()`).

Image List stores images of a fixed size and allows defining supplementary masks for drawing transparently over the background



An Image List stores the images placed in it in a device independent bitmap that is set up when you create the list. You must supply the size and estimated number of images. Considering this information, the Image List produces a corresponding bitmap. It can later be dynamically enlarged if more space is needed while the list is in use.

When the various icons, device-independent bitmaps and cursors are recorded in an Image List, they're simply copied into an area of the internal bitmap reserved for them.

An Image List can store a monochrome bitmap for each image that works as a mask, along with the RGB image itself. When you display

an image with the `ImageList_Draw()` function, the mask acts as a transparent background if the image is not supposed to cover the entire underlying surface. This is the same principle used in defining a cursor. The background is visible wherever a pixel within the monochrome bitmap is set (white).

Another capability of an Image List is the ability to save and load data from 'streams.' This allows its contents to be persistent, that is, to remain available for a subsequent run after the program ends. Finally, Image List bitmaps are very easy to use as icons for dragging objects around on the screen. The Image List is a powerful and multitasking control.

The following tables summarize the Image List functions. The functions are arranged according to the type of task performed.

Win32 functions for working with Image Lists	
Function	Task
Creating and destroying an image list	
<code>ImageList_Create</code>	Creates a new empty Image List.
<code>ImageList_Destroy</code>	Destroys an existing Image List.
Dragging with images from an image list	
<code>ImageList_BeginDrag</code>	Begins a drag operation with an image from the Image List.
<code>ImageList_DragEnter</code>	Defines a window for the drag area and displays the drag image on it.
<code>ImageList_DragLeave</code>	Hides the drag image when the user leaves the drag area.
<code>ImageList_DragMove</code>	Moves the drag image during a drag operation.
<code>ImageList_DragShowNoLock</code>	Shows or hides the drag image.
<code>ImageList_EndDrag</code>	Ends a drag operation with an image from the Image List.
<code>ImageList_GetDragImage</code>	Gets the current drag image and its position on the screen.
<code>ImageList_SetDragCursorImage</code>	Builds a new drag image for a drag operation.
Drawing images	
<code>ImageList_Draw</code>	Draws an image from the Image List to a device context.
<code>ImageList_DrawEx</code>	Extended version of <code>ImageList_Draw()</code> .
Reading and writing images in a stream	
<code>ImageList_Read</code>	Reads an Image List from a stream.
<code>ImageList_Write</code>	Writes an Image List to a stream.

Win32 functions for working with Image Lists	
Function	Task
Getting information and images from image lists	
ImageList_ExtractIcon	Takes an image from an Image List and makes it a free-standing icon.
ImageList_GetIcon	Extended version of ImageList_ExtractIcon().
ImageList_GetBkColor	Reads the current background color setting for an Image List.
ImageList_GetIconSize	Gets the height and width of an image in the Image List.
ImageList_GetImageCount	Gets the number of images in an Image List.
Adding, removing, and editing images	
ImageList_Add	Adds a bitmap as an image to an existing Image List.
ImageList_AddIcon	Adds an icon as an image to an existing Image List.
ImageList_AddMasked	Adds a bitmap as an image to an existing Image List and creates its mask.
ImageList_LoadBitmap	Creates a new Image List from a bitmap resource. (Macro for calling ImageList_LoadImage())
ImageList_LoadImage	Creates a new Image List from a bitmap, icon, or cursor resource.
ImageList_Remove	Removes an image from an Image List.
ImageList_Replace	Replaces an image in an Image List with another (bitmap).
ImageList_ReplaceIcon	Replaces an image in an Image List with an icon or cursor.
ImageList_Merge	Creates a new Image List by combining images from two Image Lists.
ImageList_SetOverlayImage	Sets one of 4 overlay images.
ImageList_SetBkColor	Sets the background color of the Image List.
ImageList_SetIconSize	Sets the size for images in the Image List and deletes any existing ones.

Creating and destroying an Image List

You might believe that to start working with an Image List, the first function you'll need is ImageList_Create(). However, InitCommonControls() is the first function needed. Although the function requires no parameters for input, nor does it return any, you cannot access any of the new common controls without it (Image Lists included). Once you have called this function, call ImageList_Create(). Here, image size and other parameters are required so the Image List can be properly configured.

```
HIMAGELIST ImageList_Create(                // Creates a new ImageList
    int    cx,                               // Width of images in pixels
    int    cy,                               // Height of images in pixels
    UINT    flags,                           // A composite of the ILC flags
    int    cInitial,                         // Number of images initially in the list
    int    cGrow                             // Increment for growth of list
);
```

The cInitial parameter is needed for determining the initial size of the reserved internal bitmap. cGrow takes effect when the list is full and another image must be added. You may, for efficiency, want to allow for several additional images to be stored. When the growth increment is exhausted, the bitmap will have to be expanded and possibly moved in memory. You don't want this to happen every time the user adds an image. Assign a higher value to cGrow if you anticipate the list growing very quickly. Otherwise assign a lower value if you don't expect much change.

Use the fFlags parameter to select the type, or internal configuration, of the bitmap — primarily the color depth of the images. The ILC_MASK flag works independently. Set this flag in addition to one of the others if a transparent display mask is desired along with the actual image.

If the call is successful, the function's return value is a handle to the newly created Image List (HIMAGELIST). All subsequent calls to Image List functions require this as input. If the create is unsuccessful, NULL is returned instead.

The handle of the Image List is also the single parameter required to delete (destroy) it. All images stored there are permanently lost when an Image List is destroyed. No further Image List functions can be called with that handle.

```

BOOL ImageList_Destroy(                                // Destroys an ImageList
    HIMAGELIST himl                                     // Handle of ImageList
);

```

The return value is TRUE if the destroy was successful, otherwise FALSE.

Loading entire bitmaps

Instead of creating an Image List, then loading individual images into it in separate steps, use two versatile functions called ImageList_LoadBitmap() and ImageList_LoadImage(). Both take a resource from a DLL or EXE file, and from it create an Image List with the images contained in the resource. First, here is ImageList_LoadImage().

```

HIMAGELIST ImageList_LoadImage(                        // Creates ImageList from resource
    HINSTANCE hi,                                     // Handle of EXE or DLL file with resource
    LPCSTR lpbmp,                                     // Ptr to string w. resource name or resource ID
    int cx,                                           // Image width
    int cGrow,                                       // Growth factor
    COLORREF crMask,                                 // Pixel color for automatic mask creation
    UINT uType,                                     // Type of resource (one of IMAGE constants)
    UINT uFlags                                     // A composite of LR constants
);

```

The function takes the instance handle of the DLL or EXE file as the first parameter. This is where the resource containing the images is found. The file must be in memory as part of the current process. The second parameter provides the resource name. You can use either a pointer to a C-string containing the resource name or the resource ID, which you cast to a string pointer with the MAKEINTRESOURCE macro.

Next the function requires the image width in pixels. Keep in mind that, although the resource can contain any number of images, they must all have the same width and height and be contiguous. By giving the function the width of one image, it can use the total width to determine the number of images (N) to be stored. The function takes the height directly from the height of the resource. The images are then transferred to the Image List and indexed sequentially from 0 to N-1.

As you saw in ImageList_Create(), the cGrow parameter is a growth factor to provide for the subsequent storing of additional images. uType contains the type of the resource being used. The table on the right shows which constants can be coded. Finally, uFlags can specify a combination of various LR constants to control image loading. Since none of these are usually needed, uFlags can be 0:

ILC constants for ImageList_Create()	
ILC_MASK	Make second bitmap for masking
ILC_COLORDBB	Store device-dependent bitmaps
ILC_COLOR4	4-bit color (16 colors), device-independent
ILC_COLOR8	8-bit color (256 colors), device-independent
ILC_COLOR16	16-bit color (65536 colors), device-independent
ILC_COLOR24	24-bit color (16 million colors), device-independent
ILC_PALETTE	8-bit color with palette, device-independent

IMAGE constants for ImageList_LoadImage()	
IMAGE_BITMAP	Bitmap
IMAGE_CURSOR	Cursor
IMAGE_ICON	Icon

LR constants for ImageList_LoadImage()	
LR_LOADFROMFILE	The lpbmp parameter does not give the resource name, but points to a string with the name of the file from which the images should be loaded.
LR_MONOCHROME	Loads images in black and white.
LR_LOADTRANSPARENT	Takes the color of the first pixel in the image, and sets all pixels of that color to the background color of the window's client area (COLOR_WINDOW)
LR_LOADMAP3DCOLORS	Gray shades previously used for 3D display RGB(128,128,128), RGB(192,192,192) and RGB(223,223,223) are changed to the system colors COLOR_3DSHADOW, COLOR_3DFACE and COLOR_3DLIGHT. I.E. Every pixel in one of the three grays is replaced by the current color for the corresponding COLOR constant.

If the load is successful, the function returns the Image List handle, otherwise NULL. You'll usually want to use this function to create an Image List from a bitmap contained in a file or resource. In this case, you can simplify the call a little by using the ImageList_LoadBitmap() macro from the Include file COMMCTRL.H.

```
#define ImageList_LoadBitmap(hi, lpbmp, cx, cGrow, crMask) \
    ImageList_LoadImage(hi, lpbmp, cx, cGrow, crMask, IMAGE_BITMAP, 0)
```

Only the first five parameters must be specified; uType and uFlags are set automatically.

Adding images

Adding images to an Image List is easy once you have one created. Use the ImageList_Add(), ImageList_AddIcon(), and ImageList_AddMasked() functions for this purpose. Which one you use depends on the situation. With ImageList_Add() you add a bitmap image using its handle to an Image List. If the color format of the image does not match that of the Image List, it will be converted. If the sizes do not match, however, the image will be rejected.

```
int ImageList_Add(                                     // Adds a new image
    HIMAGELIST himl,                                   // Handle of ImageList
    HBITMAP hbmImage,                                  // Handle of image bitmap
    HBITMAP hbmMask  // Handle of mask bitmap (masked ImageList only)
);
```

After the Image List handle, the first bitmap handle specifies the actual image to be added. The second bitmap handle is only needed if the ILC_MASK flag was on when the Image List was created. In this case, it represents a monochrome bitmap to be applied as a mask to the new image. Otherwise, specify this parameter as NULL since the function will ignore it. The result of the function is the index of the new image, that is, its sequence number within the Image List. This value remains constant and is used to identify the image for all subsequent access. If the function fails, a -1 is returned.

Once the add has successfully completed, incidentally, both bitmaps specified as input could be released without affecting the Image List. The bitmaps have been copied into the internal 'photo album.' A very similar function to the one just described is ListImage_AddMasked(). Only instead of a bitmap handle for the mask, it expects a COLORREF parameter.

```
int ImageList_AddMasked(                               // Adds bitmap as image and creates mask
    HIMAGELIST himl,                                   // Handle of ImageList
    HBITMAP hbmImage,                                  // Handle of bitmap
    COLORREF crMask                                     // COLORREF with color for mask
);
```

The function uses the color reference to generate a mask automatically for the image. For every pixel of the specified color in the image, the matching mask pixel is made transparent. The result is a type of blue-screen effect. The color, however, can be whichever one you specify with the COLORREF parameter. Choose a color that is not really needed in the image to mark those pixels where the background should show through. If the Image List does not handle masks, you can ignore the crMask

parameter and put a 0 here. Otherwise, construct the COLORREF using the familiar RGB macro, specifying three values for the color components red, green and blue (each from 0 to 255). Again for this function, a successful result returns the index of the new image, an error returns a -1.

To add an icon instead of a bitmap, use the ImageList_AddIcon() function. It takes the handle of the icon to be added but unlike ImageList_Add(), you don't need to specify a mask. This is because an icon already has its own mask. If the Image List was created with ILC_MASK set, the icon's mask is automatically stored along with the icon. Otherwise, only the icon is stored. A look at the COMMCTRL.H file shows, however, the function does not actually exist. Instead it is redirected through a macro to the ImageList_ReplaceIcon() function, which we will see in detail soon:

```
#define ImageList_AddIcon(himl, hicon) \
    ImageList_ReplaceIcon(himl, -1, hicon)
```

Again, the index of the new image is returned if the function is successful, and a -1 is returned in the case of errors. If the original icon is no longer needed, it can be released.

Drawing images

For the user to see the images you have placed in your Image List, you must draw them. The two functions ImageList_Draw() and ImageList_DrawEx() serve this purpose. The 'Ex' version, as usual, offers some extended capabilities. In many cases, however, the base version is all you'll need.

```
BOOL ImageList_Draw(                                //Draws an image
    HIMAGELIST himl,                                // Handle of ImageList
    int i,                                           // Index of image to be drawn
    HDC hdcDst,                                     // Device context where drawing should occur
    int x,                                           // X-coordinate for drawing start position
    int y,                                           // Y-coordinate for drawing start position
    UINT fStyle                                     // A composite of the ILD flags
);
```

The first two parameters identify the image to be drawn; first the Image List using its handle, then the position of the desired image within the list. Then follows the handle of the display context where the image is to be drawn. The X and Y coordinates define the drawing start position, where the upper left corner of the image will be placed. Finally, the fStyle parameter controls special display attributes that may be desired. Specify here effects such as masking, highlighting, etc.

The first two constants control the background display. They only refer to Image Lists that store masks along with the images.

The background with ILD_TRANSPARENT shows through wherever a pixel is set to 1 in the mask. For this to happen, the background must first be read and combined with the image and mask, then the result is written out to the screen. This works nicely for showing a multicolored background.

If you know, however, the background is all one color, use ILD_NORMAL instead of ILD_TRANSPARENT to make drawing faster. This eliminates reading the background, and simply fills every pixel that is set in the mask with a solid color. When this color matches the background color, the image looks transparent. Set the desired color with an Image List function called TImageList_SetBkColor() (see below). If you choose CLR_NONE as the background color, the result is like that of ILD_TRANSPARENT. Free areas of the image appear transparent, showing the background underneath.

ILD constants for ImageList_Draw() and ImageList_DrawEx()

ILD_TRANSPARENT	Makes background transparent by using mask.
ILD_NORMAL	Draws image, using the background color for all pixels defined in mask.
ILD_MASK	Draws mask without image.
ILD_IMAGE	Draws image without mask.
ILD_BLEND25	Reverses color of every fourth pixel (Raster 25).
ILD_BLEND50	Reverses color of every second pixel (Raster 50).
ILD_FOCUS	Same as ILD_BLEND25
ILD_SELECTED	Same as ILD_BLEND50

If the Image List does not contain masks, the function ignores both flags and copies the image as is to the display context. This covers the entire background. Another flag called `ILD_MASK` also lets you draw only the mask, and not the image at all. Most likely, you'll seldom find use for this in practice. The `ILD_BLEND25` and `ILD_BLEND50` constants provide ways to draw attention to selected items or controls that receive the focus. The system overlays the image with a contrasting raster of either 25 percent (every fourth pixel) or 50 percent (every second pixel). The raster pixels are drawn with inverse coloring.

Some extra features are provided by the `ImageList_DrawEx()` function. The `dx` and `dy` parameters allow you to crop the image by specifying restricted dimensions. If both values are 0, the whole image is drawn. Otherwise `dx` and `dy` give the restricted width and height, measured from the upper left corner.

[illegible]

The extended function also gives you more control over the background and foreground drawing colors. The `rgbBk` parameter specifies the background drawing color. By specifying a color to be created by the `RGB` macro, you override the background color as it was set using the `ImageList_SetBkColor()` function. To make the image transparent (assuming it has a mask) you can also give `CLR_NONE` here or to use the current background color give `CLR_DEFAULT`.

The `rgbFg` parameter is used only when either the `ILD_BLEND25` or `ILD_BLEND50` flag is set in `uFlags`. It then determines the color of the raster drawn over the image. Normally, each pixel's color is inverted to obtain the corresponding raster pixel (this is the effect produced with `CLR_DEFAULT`). However, `rgbFg` lets you generate a specific foreground color for the raster using the `RGB` macro. You've seen the background color plays an important part in displaying from image lists. Two functions here, `ImageList_SetBkColor()` and `ImageList_GetBkColor()`, let you change or check the current setting for this color.

```

COLORREF ImageList_SetBkColor(                // Set background drawing color
    HIMAGELIST himl,                          // Handle of ImageList
    COLORREF    clrBk                        // Background color (CLR_NONE for transparent)
);

COLORREF ImageList_GetBkColor(                // Returns current background color
    HIMAGELIST himl                          // Handle of ImageList
);

```

Removing and replacing images

Several functions are available for replacing, removing or otherwise changing the images within an Image List. The first we'll talk about is the `ImageList_Replace()` function. It's used to replace one image with another.

[illegible]

As in the `ImageList_Add()` function, you must supply a second bitmap handle for a mask only when working with Image Lists that have masks. The function result will be `TRUE` if the replacement was successful. The `ImageList_ReplaceIcon()` function works identically but replaces the image with an icon.

```
int ImageList_ReplaceIcon(           // Replaces an image with an icon
    HIMAGELIST himl,                // Handle of ImageList
    int i,                           // Index of image to be replaced
    HICON hicon                      // Handle of icon for new image
);
```

The function result is the index supplied by the `i` parameter, if the replace was successful, otherwise `-1`. To remove an image from an Image List, use the `ImageList_Remove()` function. Besides the Image List handle, it requires only the index `i` of the image to be removed. Assign a value of `-1` to this parameter to remove all the images without destroying the list itself.

```
int ImageList_ReplaceIcon(           // Replaces an image with an icon
    HIMAGELIST himl,                // Handle of ImageList
    int i,                           // Index of image to be replaced
    HICON hicon                      // Handle of icon for new image
);
```

If the deleted image was not the last one in the Image List, all subsequent ones are shifted up within the list's internal bitmap. This keeps the bitmap from becoming fragmented. As a consequence, the index numbers of the shifted images are reduced by 1. Your program must consider this in later references to these images. There is another way to remove all the images from an Image List. The `ImageList_SetIconSize()` function must do this when you change the image size. However, even if you reassign the same size, the images are deleted. After the function call, the Image List will be empty.

```
BOOL ImageList_SetIconSize(          // Sets image size and deletes all images
    HIMAGELIST himl,                // Handle of ImageList
    int cx,                          // New image width in pixels
    int cy,                          // New image height in pixels
);
```

Merging two images

The `ImageList_Merge()` function blends two images together to form a composite image. The two input images can come from separate Image Lists, because each is specified by its own Image List handle and index.

```
HIMAGELIST ImageList_Merge(          // Blends two images into a new IL
    HIMAGELIST himl1,                // Handle of first ImageList
    int i1,                           // Index of image in first ImageList
    HIMAGELIST himl2,                // Handle of second ImageList
    int i2,                           // Index of image in second ImageList
    int dx, // X-coordinate for origin of 2nd image relative to 1st
    int dy  // Y-coordinate for origin of 2nd image relative to 1st
);
```

The two images need not be superimposed with the same relative placement. That is, image 2 does not have to start at the upper left corner of image 1. Instead you can define an offset with the `dx` and `dy` parameters. You'll do this, for example, to center a smaller image over a larger one.

Since the composite output image can have a different size than either of its components, it is stored as the first image in a newly created Image List of its own. The function returns the handle of the new list to the caller. A return value of `NULL` indicates an error. The mask for image 2 controls its copying over image 1. Wherever a pixel was not set in image 2's mask, the contents of image 1 will show through.

Overlays

ImageList_Merge() is only one of two ways to superimpose images. You have seen how the Shell, for example, draws the overlay arrow over the shortcut icons. To blend two images for display only, without saving the composite image, call ImageList_Draw() or ImageList_DrawEx() using a special macro with the name INDEXTOOVLAYMASK. You give the macro a number from 1 to 4, to specify one of the four possible overlay images that can be defined for an Image List. The result of this macro is then added to the index of the image to be displayed, which is found in the i parameter.

A look at the macro shows the number of the overlay image is simply placed in the high-order byte of the index. (This limits the maximum number of images in a list to 256.)

```
#define INDEXTOOVLAYMASK(i)    ((i) << 8)
```

A call to ImageList_Draw() using an overlay would look something like this:

```
ImageList_Draw( hIL,                                // Draw image in DC
                iIndex + INDEXTOOVLAY(2),
                hdc,
                0,
                0,
                ILD_NORMAL );
```

With four overlay images available, you naturally need a way to distinguish them. Use the ImageList_SetOverlayImage() function for this purpose. Use it to select which images of an Image List are to serve as overlays 1 to 4.

```
BOOL ImageList_SetOverlayImage(                      // Sets one of four overlay masks
    HIMAGELIST himl,                                // Handle of ImageList
    int iImage,                                       // Index of image to use as overlay mask
    int iOverlay                                     // Index of overlay to be set (1-4)
);
```

Working with overlays is meaningful only when you are using an Image List that has masks. Since the image and its overlay are always the same size, the overlay would completely hide the image without a mask.

Getting information about images

Besides defining Image Lists, you can read from them at runtime. Therefore, not only can you get the images themselves, but you can determine how many images a list contains and various details about the bitmap representation. A function supplies each of these informational details. For example, the number of images in an Image List list is returned by the ImageList_GetImageCount() function.

```
int ImageList_GetImageCount(                          // Returns the number of images
    HIMAGELIST himl                                   // Handle of ImageList
);
```

Use ImageList_GetIconSize() to determine the size of the images. Besides the mandatory handle for the Image List you are checking, assign it the addresses of two int variables, where the function then places the image dimensions in pixels.

```
BOOL ImageList_GetIconSize(                            // Returns size of images in ImageList
    HIMAGELIST himl,                                    // Handle of ImageList
    int *cx,                                             // Pointer to int-variable for width in pixels
    int *cy                                              // Pointer to int-variable for height in pixels
);
```

Call the ImageList_GetImageInfo() function for detailed information about an image's bitmap representation. The information is returned in a variable of the IMAGEINFO type. This variable's address must be supplied in the pImageInfo parameter.

```

BOOL ImageList_GetImageInfo          // Returns information about an ImageList
HIMAGELIST himl,                      // Handle of ImageList
int i,                                // Index of image whose info is requested
IMAGEINFO *pImageInfo                // Pointer to IMAGEINFO variable
);

```

The first two fields of IMAGEINFO give you bitmap handles, one for the image and the other for its mask. With these handles you can then access the image and mask directly in the Image List. The field called IMAGEINFO.rcImage tells you where within their bitmaps the image and mask can be found. The top, left, bottom and right fields of the RECT structure gives the coordinates of the upper left and lower right corners of the image area for both bitmaps.

```

typedef struct {
    HBITMAP hbmImage;                // Handle of image bitmap
    HBITMAP hbmMask;                  // Handle of monochrome mask bitmap
    int Unused1;                      // Reserved 0
    int Unused2;                      // Reserved 0
    RECT rcImage;                    // The surrounding rectangle of both bitmaps
} IMAGEINFO;

```

So, once you have a handle for your internal photo album, you can easily get all its bitmaps. You can also use the ImageList_GetIcon() function when reading an image from an Image List. It returns an image as an icon, bringing its mask with it.

```

HICON ImageList_GetIcon(              // Creates an icon from an ImageList image
HIMAGELIST himl,                      // Handle of ImageList
int i,                                // Image index
UINT flags                            // A composite of ILD constants
);

```

The different ILD constants can be specified in flags as they were described for the two Image List Draw functions. These include, for example, ILD_FOCUS, which displays the image as the currently active screen control.

Drag and Drop with Image Lists

Because of their functionality in displaying icons, Image Lists can help you execute *Drag and Drop* operations in your programs. For example, you might want your user to have the ability to move certain controls within the application window or a dialog box. Despite help from the Image List functions, however, much work remains for your program. It must recognize the start of such an operation, set the mouse capture and move the icon across the screen by properly calling the appropriate Image List functions as it receives WM_MOUSEMOVE messages. The sections on the new TreeView and ListView controls will show you how this all looks in practice.

Here first is a theoretical overview of the different functions Image Lists provide for executing Drag and Drop operations. During a Drag and Drop operation, the user “drags” an item across the screen. Therefore, its image (called the drag image) must be dynamically redefined accordingly. The image must follow the movement of the mouse, allowing the user to position it as desired. The drag functions from the Image List API help with this. They allow you to define the image and its movements without worrying about the background ‘behind’ them. As the image shifts from its old position, the Image List functions take care of restoring the previously ‘obstructed’ screen content to its original state.

Starting a Drag and Drop operation

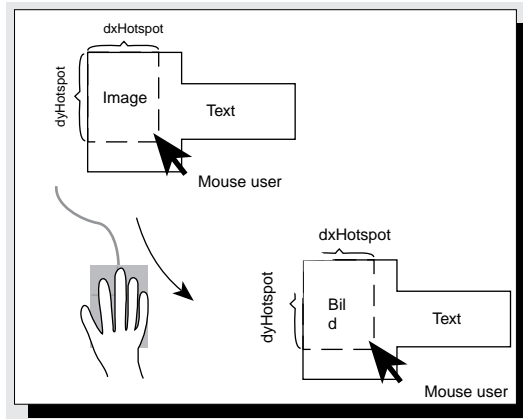
From the viewpoint of the Image List API, a new drag operation begins when a call is made to the ImageList_BeginDrag() function. At this time, the drag image is specified. It must first have been placed in an ImageList, since you use the Image List handle as the first input parameter to the function. The second parameter is the intended drag image’s index number within the list. The size of the images within the list determines the size of the drag image.

```

BOOL ImageList_BeginDrag(           // Start dragging an image from the ImageList
    HIMAGELIST himl,                // Handle of ImageList
    int iTrack,                      // Image index
    int dxHotspot,                  // X-offset of dragpoint from top left
                                   // image corner
    int dyHotspot                   // Same as above for Y-offset (both in pixels)
);

```

Fixing the hotspot offset gives the mouse cursor a tight hold on the image



The dxHotspot and dyHotspot parameters define the offset between the cursor's hotspot and the upper left corner of the image. The coordinates 0/0 indicate the point of the mouse cursor is directly over this corner. The user usually has initiated the drag after placing the pointer somewhere else inside the image. Maintain the same offset throughout the operation so the movement of the image exactly follows that of the mouse.

To give the mouse cursor a tight hold on the image, compute the coordinates of the hotspot as pixel offsets before calling ImageList_BeginDrag(). Then pass these values to it in the dxHotspot and dyHotspot parameters.

The ImageList_BeginDrag() function by itself does not yet make the image visible. Next, you must call ImageList_DragEnter(). Its primary purpose is to define the area within which the image is allowed to be displayed. You can make this the current control, a dialog box, or even the entire screen, depending on your needs. To limit the dragging range, pass a window handle for the restricted area to the function. If the user drags the image beyond the boundaries, it disappears. To allow dragging anywhere on the desktop, give this parameter a value of 0.

```

BOOL ImageList_DragEnter(           //Limits drag area to specified window
    HWND hwndLock,                 // Handle of window or 0 for desktop
    int x,                         // X coordinate of image start position
    int y                          // Y coordinate of image start position
);

```

As the x and y parameters, the function needs the initial coordinates of the image relative to the upper left corner of the specified window.

Moving the drag image

Once the operation is started, you must make the image follow the cursor as the user moves the mouse. To accomplish this, call the ImageList_DragMove() function. Pass the new X and Y coordinates of the mouse to the ImageList_DragMove() function. You'll receive these in a WM_MOUSEMOVE message. If the window receiving the message is not the one identified in ImageList_DragEnter(), you must first convert the coordinates so ImageList_DragMove() can interpret them correctly for the appropriate window. Mouse movement then automatically occurs with respect to the mouse hotspot defined for ImageList_BeginDrag().

```

BOOL ImageList_DragMove(           // Moves the drag image to a new position
    int x,                         // New X and Y coordinates for drag image
    int y                          // relative to window specified in DragEnter
);

```

To highlight a screen control as a potential drop target when the mouse and drag image move over it, temporarily hide and then redisplay the drag image. The `ImageList_DragShowNoLock()` function can be used to produce this effect.

```

BOOL ImageList_DragShowNoLock(           // Show or hide the drag image
    BOOL fShow                          // TRUE = show, FALSE = hide
);

```

To determine the current position of a drag image or hotspot during a drag operation, use the `ImageList_GetDragImage()` function. This function expects two pointers to `POINT` structures. It fills one with the current coordinates of the upper left corner of the drag image, the other with those of the hotspot. Both are expressed relative to the upper left corner of the window defined as the drag area in `ImageList_DragEnter()`. As the function result, the handle of the Image List is also returned.

```

HIMAGELIST ImageList_GetDragImage(       // Returns current position of drag
                                         // image
    POINT *ppt,                          // Pointer to POINT var for position of drag image
    POINT *pptHotspot                   // Pointer to POINT var for position of hotspot
);

```

Leaving a window

Call `ImageList_DragLeave()` when the drag operation ends or when an attempt is made to leave the valid drag area. The function simply needs the handle of the window previously specified for `ImageList_DragEnter` as input. When you call the function, the drag image disappears.

```

BOOL ImageList_DragLeave(                 // Indicates leaving the drag-area window
    HWND hwndLock                        // Window handle
);

```

Ending the drag operation

The `ImageList_EndDrag()` function works as the opposite of `ImageList_BeginDrag()` to end a drag operation. No parameters are required.

```

void ImageList_EndDrag( );               // Ends current drag operation

```

Reading and writing in streams

Sometimes you'll want to save an Image List that your program has created for use in a subsequent run. This is a typical requirement when working with objects that are to be made persistent, so that their contents are maintained after the program ends. OLE defines and implements the concept of "structured storage" for this purpose. It allows you to divide a file into separate areas (storages), somewhat similar to subdirectories.

A storage can contain any number of streams into which OLE objects place their data to become persistent. The OLE object must support two interfaces called `IStream` and `IStorage` defined for this purpose. The interfaces provide a standard way for objects to pass their contents back and forth to files.

`ImageList_Read()` reads an Image List from a stream, into which it was previously written by `ImageList_Write()`. To run `ImageList_Read()`, you need a pointer to an `IStream` interface, which represents the stream to be read within a file.

```

HIMAGELIST ImageList_Read(               // Reads an ImageList from a stream
    LPSTREAM pstm                       // Pointer to IStream interface
);

```

If the data was successfully read, the function returns the handle of the Image List that was loaded, and you can now use the handle to call any of the normal IL functions. The following function called `LoadImageList()` is our example, taken from the

end of the chapter with IMAGEDIT. It contains all the necessary OLE calls to first obtain a stream interface, and then read the Image List saved there by calling `ImageList_Read()`.

The function expects only two string pointers as arguments. The first points to a string with the storage name, in this case, the name of the file where the Image List was previously written. A storage can contain multiple streams, so the second pointer references a string with the stream name, which can contain up to 31 characters.

```
HIMAGELIST ImageList_Read(           // Reads an ImageList from a stream
    LPSTREAM  pstm                   // Pointer to IStream interface
);
```

As the counterpart to `ImageList_Read()`, the `ImageList_Write()` function writes an Image List into a stream. Besides the Image List handle, it needs a pointer to an `IStream` interface. This identifies the stream where the data is to be written. A result of `TRUE` (OK) or `FALSE` (error) is returned.

```
HIMAGELIST ImageList_Read(           // Reads an ImageList from a stream
    LPSTREAM  pstm                   // Pointer to IStream interface
);
```

We have provided a function to help you use `ImageList_Write()` also. Named `SaveImageList()`, it expects the handle of the Image List and two string pointers. One string names the storage (file) where the data should be written, the other names the stream. By assigning different stream names, you can write streams from different Image Lists to the same storage. The unique names allow you to use `LoadImageList()` later to read whichever Image List you want from the file.

```
HIMAGELIST ImageList_Read(           // Reads an ImageList from a stream
    LPSTREAM  pstm                   // Pointer to IStream interface
);
```

The system Image List

Even if you have not created an Image List of your own, two are available in the system that you can access at any time. These are where the desktop keeps all the file and folder icons that it has so far displayed. One is for large icons and the other for small icons. When you have their handles, you can manipulate these with the IL functions, just the same as Image Lists you create. Of course, as always when you access the system's own data, you want to be careful not to interfere with their intended use.

The following partial code shows a call to the `SHGetFileInfo()` function from the Shell API. This function gives you the handles of the system Image Lists. The handle requested here is the one for the large icons, represented by the `SHGFI_ICON` constant. If `SHGFI_SMALLICON` were specified instead, the handle of the small icon Image List would be returned.

```
g_hImagesNormal= (HIMAGELIST)SHGetFileInfo( "",
                                           0,
                                           &sfi,
                                           sizeof( sfi ),
                                           SHGFI_SYSICONINDEX |
                                           SHGFI_ICON );
```

For more information on how this function works, see Chapter 43 on the Shell.

*The key for the
Windows 95
controls*

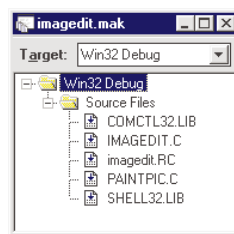


Using Image Lists

The IMAGEDIT program shows how to use the various Image List functions in more detail. The program manages an Image List with an image size of 32x32 pixels, which are shown in a listbox control. There are two ways to place images into the Image List. The first is to use the button labeled 'Add system images'. Click this button to have the program scan the system Image List containing the standard icons and copy them one by one to its own Image List. This is a quick way to look at the complete system Image List contents. The images can be copied "as is" because they maintain their size of 32*32 pixels.

This is usually not true for images added by the second method of placing images into the Image List. These are bitmap (.BMP) files displayed on the Desktop or in the Explorer, which you can simply drag into the listbox. The listbox checks the name of the file using a standard mechanism described in the Shell chapter. If it is in fact a bitmap file (it may also be a DIB file), the image is shrunk to 32*32 and added to the end of the program's Image List. The image then appears in the listbox.

*The MAKE file
for the sample
program
IMAGEDIT*



The program works like a mini slide viewer for bitmap files that are stored on disk. By noting the file names as it reads the images, the program can take the user right to the source file for any that is clicked.

You've already seen the SaveImageList() and LoadImageList() functions. The program uses these to save and load the Image List the user constructs. When the program ends, it automatically saves the Image List in the IMAGE.DAT file under the current directory. It opens this file again and loads the data from it the next time it runs.

**You'll find the following program(s) on
the companion CD-ROM**



IMAGEDIT.C (C listing)

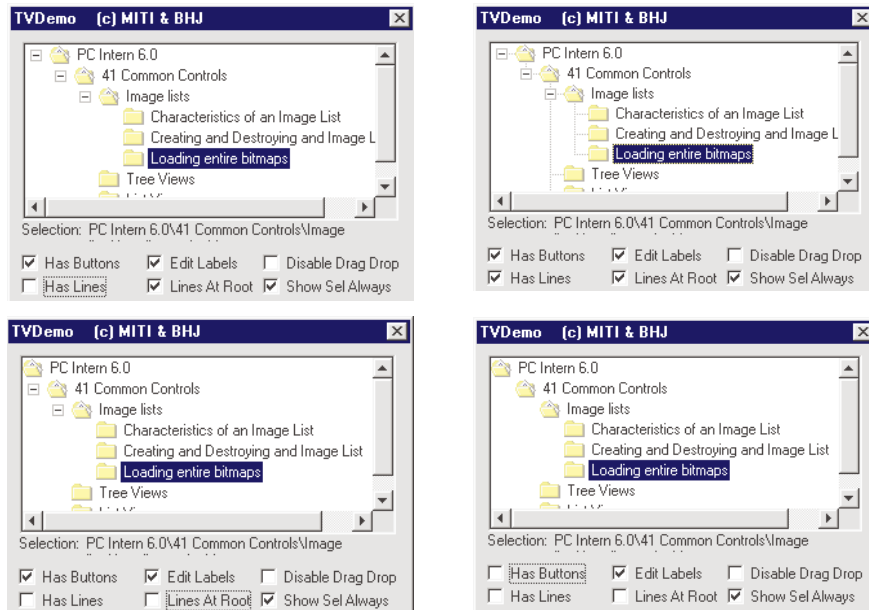
The program also shows the different ways an image can be drawn with the ImageList_Draw function. When an image in the listbox is clicked, four variants of it appear beneath the listbox: the standard image, the two dithered versions (using the ILD_BLEND25 and ILD_BLEND50 options), and the mask.

Finally, there is a managed Drag and Drop capability inside the dialog box. Any image can be dragged from the listbox and dropped into the trash can. This removes it from the internal Image List. The following listing shows you how it all looks.

Tree Views

The TreeView control is another of the new common controls of Windows 95. The Explorer's appearance comes from both the Tree View and ListView controls. TreeViews are the ideal means for displaying hierarchical arrangements of all types. Use TreeViews whether you want to list the chapters of a book or catalog, a manufacturer's products or the directory tree of a hard drive. In other words, use TreeViews any place where several other (child controls) can arise from one (parent) control. You can, of course, build such controls yourself or purchase them from third-party vendors. The TreeView is well conceived, offers several options and holds open a few doors for programmers, allowing them to maintain control over the actions of the user and how the tree is represented.

Different views of a Tree View



Communication with TreeViews

TreeViews, like most other common window controls, create windows with their own window class and their own messages by which they are controlled. Their class carries the constant name `WC_TREEVIEW`. It's broken down within the corresponding Include file `COMMCTRLS.H` into the string "SysTreeView32" during compilation for Windows 95. This file contains all types, constants and declarations needed to manage such a control.

The messages of an application to a TreeView control use the prefix `TVM` (TreeView Message). They're sent to a TreeView control over the normal pathways using `SendMessage()`. However, let's first look at the various `TVM` messages, which will already provide some idea of the capabilities of the control.

The table on the next page shows the following: To make access to a TreeView more understandable and to avoid always calling `SendMessage()` explicitly, a corresponding macro is defined in the Include file `COMMCTRL.H` for each `TVM` message. It allows the message transmission to appear like a function call. It's always quite easy to determine the name of the macro from that of the respective `TVM` message simply by removing the `TVM` prefix of the message name.

Then prefix the result with "TreeView" and adjust the use of uppercase/lowercase. So, the `TVM_GETCOUNT` message, used by a caller to determine the number of items in a TreeView, becomes `TreeView_GetCount`. All these macros expect the window handle as the first argument. No message can be sent without this window handle. `TreeView_GetCount()`, requiring no additional parameters, returns the number of items received from `SendMessage()` as the function result. The following defines the macro from `COMMCTRL.H`.

```
#define TreeView_GetCount(hwnd) \
    (UINT)SendMessage((hwnd), TVM_GETCOUNT, 0, 0)
```

Other function macros, whose corresponding `TVM` messages require several parameters, are somewhat more complex. Fortunately, they're easy to handle if you know the meaning of the `wparam` and `lparam` of the message in question. The following example is `TreeView_SortChildrenCB()`, where it is necessary to provide two additional parameters besides `hwnd`.

```
#define TreeView_SortChildrenCB(hwnd, psort, recurse) \
    (BOOL)SendMessage((hwnd), TVM_SORTCHILDRENCB, \
        (WPARAM)recurse, (LPARAM)(LPTV_SORTCB)(psort))
```

TreeView controls are managed by sending TVM messages (TreeView Messages)		
TVM Message	Corresponding macro	Task
TVM_INSERTITEM	TreeView_InsertItem()	Inserts a new item
TVM_DELETEITEM	TreeView_DeleteItem()	Deletes items.
TVM_SETITEM	TreeView_SetItem()	Sets the attributes of an item.
TVM_GETITEM	TreeView_GetItem()	Queries the attributes of an item.
TVM_SETIMAGELIST	TreeView_SetImageList()	Links the control to an image list.
TVM_GETIMAGELIST	TreeView_GetImageList()	Returns the handle of the set image list.
TVM_EXPAND	TreeView_Expand()	Makes parent items larger or smaller.
TVM_SETINDENT	TreeView_SetIndent()	Set indentation of the child levels.
TVM_GETINDENT	TreeView_GetInden()	Query indentation of the child levels.
TVM_GETCOUNT	TreeView_GetCount()	Query number of items in the control.
TVM_GETNEXTITEM	diverse macros	Run through the items of a TreeView.
TVM_SELECTITEM	TreeView_SelectItem()	Determines the item selected.
TVM_ENSUREVISIBLE	TreeView_EnsureVisible()	Makes sure that a certain item is visible in the client area of the control.
TVM_GETVISIBLECOUNT	TreeView_GetVisibleCount()	Returns the number of items visible in the client area of the control.
TVM_GETITEMRECT	TreeView_GetItemRect()	Returns the drawing area of an item.
TVM_HITTEST	TreeView_HitTest()	Assigns a visible item to a coordinate.
TVM_EDITLABEL	TreeView_EditLabel()	Sets the editing of an item label into motion.
TVM_GETEDITCONTROL	TreeView_GetEditControl()	Returns the window handle of the edit control to be used for editing.
TVM_ENDEDITLABELNOW	TreeView_EndEditLabelNow()	Forces the end of editing.
TVM_SORTCHILDREN	TreeView_SortChildren()	Ascending alphanumeric sort.
TVM_SORTCHILDRENCB	TreeView_SortChildrenCB()	Sort with a sorting function prepared by the caller.

We'll always introduce the macros along with the messages in this chapter to make your work easier. It's up to you whether you prefer using the macros or the direct calling of `SendMessage()`; identical program code is generated on the lowermost level in either case.

TreeView notifications

TreeView notifications are the control's response to the user's actions or to TVM messages sent by a program. They're passed to the window designated as the parent window when the control was generated, for example, the message procedure of a dialog box. As a result, the window gets the opportunity to respond to the event. The following table on the following page summarizes the different notifications beginning with the prefix TVN.

The parent always receives the TreeView notifications in the context of a `WM_NOTIFY` message. Therefore, their content need only be analyzed initially to discover the TreeView notification. You will find the control ID of the respective TreeView control in the `wParam` of the received message. `lParam` contains a pointer to the a structure with additional information about the events which led to the sending of the message. `lParam` refers to various data structures depending upon the notification. All the data structures share at least one common item: a structure of the `NMHDR` type is positioned at the beginning. It again includes the sender of the message and, most importantly, the notification itself.


```

        if( ptvdi->item.pszText )
        {
            // Yes, set new item text
            ptvdi->item.mask = TVIF_TEXT;
            TreeView_SetItem( hTreeView, &ptvdi->item );
        }
    }
    break;
}
break;
}
break;
}
}

```

Generating a TreeView during runtime

A TreeView is generated during runtime by `CreateWindowsEx()` with the specification of the TreeView window class. This is followed by the traditional pattern according to which other controls are also generated. First, however, a preceding call is required to the `InitCommonControls()` function from the `COMCTL32.DLL`. This is to implement the common controls. The TreeView window class can only be set up by calling this function; otherwise it would not be possible to create the control. Enter the creation attributes for the TreeView along with the usual WS constants (ws = Window Style) in the `dwStyle` parameter of the `CreateWindowEx()` function. They're represented by six TVS constants which determine the appearance of the created TreeView control and the way it operates.

The components of a TreeView item

Along with the label, each item of a TreeView has an icon available, which is inserted from a connected image list. You don't necessarily have to work with icons, although this is an option. Moreover, including `TVS_HASBUTTONS` in the `dwStyle` parameter guarantees that a plus or minus sign will automatically appear next to all TreeView items containing child items. A plus sign will appear if the item is not opened and a mouse click on the plus sign will render the child items visible. The plus sign then becomes a minus sign which can be used to close the parent item again.

To attach the individual items to a line, add `TVS_HASLINES`. Without the additional entry of `TVS_LINESATROOT`, the highest item of a TreeView will remain the item at the root of the tree, spared from a connection to its "underworld". It can also be used as a type of header. However, with `TVS_LINESATROOT`, the root item is handled like any other item.

TVS constants (TreeView Styles) determine the appearance and the mode of operation of a TreeView control.

TVS_HASBUTTONS	A plus-minus button will appear next to each item, if it has child entries.
TVS_HASLINES	The individual items are connected by lines.
TVS_LINESATROOT	A line also exits from the root of the tree.
TVS_EDITLABELS	The user is able to edit an item's label.
TVS_SHOWSELALWAYS	Ensures the selected item will be visible, even if the control does not have the focus.
TVS_DISABLEDRAHDROP	No drag and drop support.

The TreeView control lets the user edit the label of the various items (you're probably familiar with this from the Explorer). If the user clicks an item an instant longer than normal, the TreeView will open an Edit control over the item, in which the existing label is displayed. Make certain to specify `TVS_EDITLABELS` in the window style.

While the other TVS constants switch on a specific attribute, `TVS_DISABLEDRAHDROP` does just the opposite: It switches off detection of a drag operation by the control, which, by default is switched on. The user is then unable to remove any items

from the tree using Drag and Drop. The various styles have some implications for the messages which the TreeView control processes at runtime, and thus also for the notifications it sends to its parent. We'll talk about this again later in this chapter.

Switching between the various styles

As with the other controls, you need not settle upon the final appearance of a TreeView control when you generate it. You can change this setting later and even adapt the appearance of the control to new conditions. The well known API functions, `GetWindowLong()` and `SetWindowLong()`, return the current window style or set it.

You can change a whole series of settings which Windows maintains for each window using the API function, `SetWindowLong()`, from the `USER32.DLL`. For example, you can change the address of the message procedure or even the current window style. The window style is represented by the constant `GWL_STYLE` which must be entered as a second parameter when calling `SetWindowLong()`.

```
LONG SetWindowLong(
    HWND hWnd,                // Handle of the window addressed
    int nIndex,                // Offset of the LONG, one of the GWL constants
    LONG dwNewLong             // new value
);
```

Besides that, the function expects the window handle of the window addressed in the first parameter and the new value for the window style in the last parameter. Because you only want to change a specific flag in the window style and leave all the others as they are, first query the current window style. Insert or remove the desired flags and then call `SetWindowLong()` to set the new style in the window.

Use the API function `GetWindowLong()` to get the window style. This function also expects the window handle and the offset of the desired LONG as the GWL constant for the argument. In our case, this is, again, `GWL_STYLE`. Then the current window style is returned as the function result.

```
LONG GetWindowLong(
    HWND hWnd,                // Window handle
    int nIndex                 // One of the GWL constants
);
```

The following code sequence shows how a TreeView control is first generated. Then the editing option of the individual items is switched on and off with `SetWindowLong()` and `GetWindowLong()`.

```
STYLESTRUCT st;
// Get old style
st.styleOld = GetWindowLong( hTreeView, GWL_STYLE );
st.styleNew = st.styleOld;
if( st.styleNew & TVS_EDITLABELS )
    st.styleNew &= ~TVS_EDITLABELS; // Remove
else
    st.styleNew |= TVS_EDITLABELS; // insert
// Set new style
SetWindowLong( hTreeView, GWL_STYLE, st.styleNew );

// And notify control of change
SendMessage( hTreeView,
    WM_STYLECHANGED,
    ( WPARAM ) GWL_STYLE,
    ( LPARAM ) &st );
```

Adding and deleting items

Use the TVM_INSERTITEM message to insert new items in a TreeView control. This message is mapped to the `TreeView_InsertItem()` macro. The macro expects a 0 in `wParam` while it expects the address of a structure of the `TV_INSERTSTRUCT` type in `lParam`. The item to be inserted is described in `TV_INSERTSTRUCT`.

```
HTREEITEM TreeView_InsertItem( // Macro for TVM_INSERTITEM, inserts element
    HWND          hwnd,        // Handle of the TreeView
    LPTV_INSERTSTRUCT lpis      // Pointer to TV_INSERTSTRUCT structure
);
```

Notice the `HTREEITEM` type which you'll often encounter when accessing TreeViews. `HTREEITEM` represents the handle of an item within a TreeView, in other words, a "tree item." Each item receives its characteristic `TreeItem` handle when it's inserted into a TreeView. This handle accompanies the item throughout its entire life and identifies it. All TVM messages that act upon a given item expect as a parameter the `TreeItem` handle of the addressed item.

The `HTREEITEM` item is defined in the Include file `COMMCTRL.H` as a pointer to a structure of the `_TREEITEM` type. The structure of `_TREEITEM` is not known, but it represents the individual items within a TreeView.

```
typedef struct _TREEITEM FAR* HTREEITEM;
```

What is known is that the TreeView dynamically creates the memory for new items using the heap, so the maximum number of items is limited only by the amount of free disk space on the heap. Each item of the `_TREEITEM` type therefore requires 64 bytes.*

Two `TreeItem` handles are likewise found in `TV_INSERTSTRUCT`. It's needed to insert a new item:

```
typedef struct {
    HTREEITEM hParent;                // Handle of the present element
    HTREEITEM hInsertAfter;          // Handle of the predecessor within the child
    TV_ITEM item;                    // The new item
} TV_INSERTSTRUCT, FAR *LPTV_INSERTSTRUCT;
```

The two handles determine where the new item is inserted in the TreeView. (A TreeView does not need to be constructed from top to bottom as a rigid sequence). At this point, new items can also be inserted later deep within the hierarchy. For that purpose, the `TV_INSERTSTRUCT` structure requires the `TreeItem` handle of the parent in the `hParent` field, in other words, of the superordinate item. If the `TVI_ROOT` constant or `NULL` is specified here, the item is accepted into the TreeView as a new root and therefore precedes all the items already inserted.

The second `HTREEITEM` in the `hInsertAfter` field describes where the new item is to be located under the parent item. Finally, since each parent can have a series of daughter items, the question arises where to sort the new item (beginning, end or in the middle). If the new item is to be intentionally sorted behind an already existing one, whose `TreeItem` handle is known, simply enter this handle in `TV_INSERTSTRUCT.hInsertAfter`. If you don't have an available `TreeItem` handle for the predecessor, specify one of the three following TVI constants instead.

TVI constants (TreeView Insertstruct) determine the position assignment of a new item beneath its parent

Constant	Task
TVI_FIRST	Accept new item into the list as the first in the list of child items.
TVI_LAST	Attach new item to the end of the list.
TVI_SORT	Sort new item into the list alphabetically

That's how you determine the position of the new item within the existing hierarchy. The `TV_ITEM` structure, which must be entered in the `TV_INSERTSTRUCT.item` field, describes the new item. Here, for example, the label and the icon of the item as well as its attributes are established.

The TV_ITEM data structure

However, the TV_ITEM data structure is not used only with the TVM_INSERTITEM message. Many other messages use its services not only for receiving information but also for returns. The TVM_GETITEM message (which we'll talk about soon) is one example. It returns information on a specified item to the caller in a TV_ITEM structure sent by the caller.

```
typedef struct {
    UINT          mask;                // Describes item to be inserted
    HTREEITEM     hItem;              // A combination of the TVIF constants
    // TreeItem handle of the item described by the
    // structure.
    UINT          state;              // Current state, combination of the TVIS constants.
    UINT          stateMask;          // Possible states, combination of the TVIS
    // constants
    LPSTR         pszText;            // Pointer to label
    int           cchTextMax;         // Size of the buffer
    int           iImage;             // Number of the standard icon from the image list
    int           iSelectedImage;     // Number of the icon in the selected state
    int           cChildren;          // 1, if the item has children, otherwise 0
    LPARAM        lParam;             // DWORD which is linked to the item
} TV_ITEM, FAR *LPTV_ITEM;
```

Because the structure is used in very different contexts, the caller doesn't have to initialize all the fields. Consequently, in the first field, mask, a combination of the TVIF constants is listed. These constants provide information on which fields are filled with valid data. A caller sets this mask to inform the message of which fields it has initialized. If the structure for returning information is used by a TreeView notification, you'll find out which fields are initialized with valid information from the response.

TVIF constants (TreeView Item Flags) indicate which fields of the TV_INSERTSTRUCT are initialized.	
Constant	Following fields are initialized ...
TVIF_TEXT	pszText and possibly also cchTextMax
TVIF_STATE	state and stateMask
TVIF_IMAGE	iImage
TVIF_SELECTEDIMAGE	iSelectedImage
TVIF_PARAM	lParam
TVIF_CHILDREN	cChildren

hItem

Usually, you use hItem to designate the addressed item by specifying its TreeItem handle. The only exception is when the item is created using TVM_INSERTITEM, since it doesn't yet have such a handle. The handle will be returned as the result of the SendMessage() call. Therefore, you can ignore this field (at least in this case).

state and stateMask

The state field describes the current status (state) of an item through a combination of the TVIS constants. Use the stateMask field to indicate which of the flags is to be changed or set. Use the same TVIS constants as in state. For example, to display an item in bold, set the TVIS_BOLD constant in state. However, to ensure that this constant is noticed, insert it at the same time in stateMask.

It also works the other way around: If the TVIS_BOLD flag is set in stateMask, the control assumes the new status desired for this attribute (in this case: bold text, yes or no) is set in state. However, if the TVIS_BOLD flag has not even been set in state, it means that from now on the item is no longer to appear in bold. This is how the two fields work together.

TVIS constants (TreeView Item Style) describe a TreeView item.	
Constant	Meaning
TVIS_SELECTED	The item is currently selected.
TVIS_DISABLED	The item cannot be selected.
TVIS_EXPANDED	The item is opened so its child items are visible.
TVIS_BOLD	The label of the item appears in bold.
TVIS_CUT	The item is currently being cut or pasted.
TVIS_DROPHILITED	The item is highlighted as the target of a drag and drop operation.
TVIS_OVERLAYMASK	A different image from the image list is to be overlaid over the item's icon item (an overlay image).
TVIS_STATEIMAGEMASK	A state icon is to be shown next to the item's icon.
TVIS_EXPANDEDONCE	No TVN notifications are to be sent to the parent window when the item is being opened or closed.

The meaning of the constants will become more clear. However, keep in mind that besides the bold attribute (TVIS_BOLD), it's unfortunately impossible to assign any additional font attribute to an item. Also, it's not possible to make custom font settings for some or all of the items. As a rule, when inserting new items, limit yourself to specifying 0 for both fields. This lets you avoid setting either of these flags.

pszText and cchTextMax

These two fields are used to transmit the label the item presents to the user. An application, if supplied with this label, doesn't need to initialize cchTextMax and can restrict itself to entering a pointer to the C-string with the desired label in pszText. The TreeView uses that information to determine the length automatically. However, if pszText is used to receive information from the TreeView, cchTextMax must be initialized with the size of the buffer to which pszText points, before the message is sent. As a result, the TreeView can make certain the buffer is large enough before depositing a string in this buffer and it won't accidentally overwrite the buffer with a label which is too long for it.

ilImage and iSelectedImage

This is where the icons that appear next to the item are determined. They refer to the image list to which TreeView is linked using the TVM_SETIMAGELIST message (this is explained in a later section). ilImage specifies the index of the icon in normal state and iSelectedImage specifies the index of the icon in the selected state. If desired, use the same value for both to have the control display the same icon in both states.

cChildren

This flag indicates if the item has subordinate items, i.e., children. 0 stands for "childless." Any other value stands for "parent."

IParam

An application uses this long value to place its own information in an item, for example, a pointer to a program internal data structure that describes the content and the meaning of the item in greater detail. If the individual items are to represent objects of the application, a connection can be produced in this way between the object instance and its visual equal in the TreeView.

Adding items in practice

At first glance, an overwhelming number of structures, fields and constants are initialized when a Tree View is generated and initialized. Fortunately, the chaos becomes clearer when you've carefully sifted through the corresponding code. The following partial program code first generates a new TreeView and then inserts two items with the names "Alphabet forward" and "Alphabet backward" below the root. Then 26 child items with the letters A through Z are generated below each of these two items, once forward, once backward.

```

TV_INSERTSTRUCT tvis;           // Structure for the insertion of TreeItems
HTREEITEM        hVor,          // Handles for Alphabet forward and backward
                  hBack;

char              Letters[ 2 ]; // Text buffer
int               i;

tvis.hInsertAfter = TVI_LAST;    // Insert new entry at end
tvis.item.mask = TVIF_TEXT;     // Set flags for new item

// Generate parent item -----
tvis.item.pszText      = "Alphabet forward"           // Set label
tvis.item.cchTextMax   = lstrlen(tvis.item.pszText );
tvis.hParent           = TVI_ROOT;
hVor = TreeView_InsertItem( hTreeView, &tvis );       // Insert entry

tvis.item.pszText      = "Alphabet backward"          // Set label
tvis.item.cchTextMax   = lstrlen(tvis.item.pszText );
tvis.hParent           = TVI_ROOT;
hBack = TreeView_InsertItem( hTreeView, &tvis );      // Insert item

Letters[ 1 ] = '\0';

// Insert letters of the alphabet as items -----
for( i = 0; i < 26; i++ )
{
    Letters[ 0 ] = 'A' + i;
    tvis.item.pszText      = Letters;                 // Set label
    tvis.item.cchTextMax   = 1
    tvis.hParent           = hVor;                    // Below alphabet forward
    TreeView_InsertItem(hTreeView, &tvis);            // Insert item

    Letters[ 0 ] = 'Z' - i;
    tvis.item.pszText      = Letters;                 // Set label
    tvis.item.cchTextMax   = 1
    tvis.hParent           = hBack;                   // Below alphabet backward
    TreeView_InsertItem(hTreeView, &tvis);            // Insert item
}

```

Deleting items

Once an item is generated, you can also quickly delete it using the TVM_DELETEITEM message. To do this, specify 0 in wParam and the TreeItem handle of the item to be deleted in lParam. As a result, a value of TRUE is returned if it was possible to delete the item, otherwise FALSE is returned. Simply pass the TreeItem handle of the item to be deleted in addition to the window handle of the TreeView to the corresponding macro function named TreeView_DeleteItem().

```

#define Header_DeleteItem(hwndHD, i) \
    (BOOL)SendMessage( (hwndHD), HDM_DELETEITEM, (LPARAM)(i), 0L)

```

Also, a function macro named TreeView_DeleteAllItems() is available that deletes all the items of a TreeView. As the following partial program listing from the Include file COMMCTRL.H shows, the TreeView is sent a TVM_DELETEITEM message for that purpose. The TVI_ROOT constant, which describes the root item, is specified as the handle of the TreeView item to be deleted. Because the root item is deleted, the entire contents of the tree are deleted.

```

#define TreeView_DeleteAllItems(hwnd) \

```

```
(BOOL)SendMessage( (hwnd), TVM_DELETEITEM, 0, (LPARAM)TVI_ROOT)
```

The partial program listing highlights the operating method of the TVM_DELETEITEM message. It automatically deletes all child items of an item given to it for deletion. So there is no need to run through each branch completely and delete each item with a separate TVM_DELETEITEM message. It's sufficient to delete the parent of the branch and all subsequent items of the branch will be removed.

No deletion without notification

The TreeView sends its parent window various notifications. One of these notifications is the TVN_DELETEITEM message. It notifies the parent window that an item in the TreeView has just been deleted. The parent doesn't have to concern itself with it and can ignore this message if it was only a matter of deleting the item.

However, an active response to this message may be desirable, for example, if you create an item and allocate additional memory for a custom data structure using the heap and this is linked to the item by the TV_ITEM.IParam field. The corresponding memory must now be released again. Otherwise you will lose the only reference to it, leaving the memory on the heap as a dead zone.

If you are deleting only a single item by sending a TVM_DELETEITEM message, you'll usually be acquainted with the item, so you can release the memory again beforehand. However, what if you cut down the entire tree or at least "saw off" a relatively large branch? Does that mean you have to go through all the items below it manually before sending the decisive TVM_DELETEITEM message to get the IParam from the TV_ITEM structure of each item to release the memory allocated for the item? The answer is no, that's precisely what the DELETEITEM notification is there for.

The TreeView generates a TVN_DELETEITEM message to the parent window for each item it deletes. It doesn't matter whether a TVN_DELETEITEM message deletes a single item or hundreds. A TVN_DELETEITEM message will be sent to the parent for each item, accompanied by the item's TV_ITEM structure. As a result, the parent can infer the IParam of the item from this structure and release the memory to which it is pointing.

In concrete terms, the IParam received with the TVN_DELETEITEM points to a structure of the NM_TREEVIEW type, which is used with an entire series of TreeView notifications.

```
typedef struct {                // Describes TreeView item during notification
    NMHDR    hdr;                // Message header with sender and TVN code
    UINT     action;             // with different meanings according to the message
    TV_ITEM  itemOld;            // TV_ITEM structure with old item
    TV_ITEM  itemNew;            // TV_ITEM structure with new item
    POINT    ptDrag;             // Mouse coordinates for drag operation
} NM_TREEVIEW, FAR *LPNM_TREEVIEW;
```

When a TVN_DELETEITEM message is received, it's always the case that only the itemOld field is filled. It contains a TV_ITEM describing the item just deleted with all its data.

The following code shows how to delete the currently highlighted item and, upon receiving a TVN_DELETEITEM message, release the memory which was dynamically generated when the item was created. Because you placed the pointer in the IParam of the TV_ITEM structure, this is not a problem.

```
HITEM hItem;

// Get current item _____
hItem = TreeView_GetSelection( hTreeView );
if( hItem )                                // and delete if present
    TreeView_DeleteItem( hTreeView, hItem );

...
...
```

```

...

// - Message procedure of the parent window -----
switch( wParam )
{
    case WM_NOTIFY:
        switch( LOWORD( wParam ) )
        {
            case IDC_TREEVIEW:                // Notification of TreeView control
            {
                LPNM_TREEVIEW pnmTV = (LPNM_TREEVIEW)lp;
                switch( pnmTV->hdr.code )
                {
                    case TVN_DELETEITEM:        // Deletion notification

                        // Release the memory space allocated with malloc()
                        // again
                        if( pnmTV->itemOld.lParam )
                            free( ( LPVOID )pnmTV->itemOld.lParam );
                        break;
                }
            }
            break;
        }
    break;
}
}

```

Querying and setting items

If existing items are to be changed at runtime, use the TVM_SETITEM message or the corresponding function macro `TreeView_SetItem()`. The TreeView control expects a 0 in the wParam for this message, while in the lParam it expects the address of a TV_ITEM structure with the item's data.

```

#define TreeView_SetItem(hwnd, pItem) \
    (BOOL)SendMessage( (hwnd), TVM_SETITEM, 0, \
        (LPARAM)(const TV_ITEM FAR*)(pItem) )

```

Before calling the function, load the handle of the TreeView item whose settings are to be changed by the call in the TV_ITEM.hItem field. Use the mask field to inform the TreeView which settings are to be changed. In this field, each setting to be changed is represented by one of the TVIF constants already introduced. For example, specify TVIF_TEXT | TVIF_IMAGE to simultaneously change the item's label and the number of the icon for non-selected display. The TreeView loads the new settings from the corresponding fields in TV_ITEM, in this case, from TV_ITEM.pszText (the label) and TV_ITEM.iImage (the icon number).

To change one of the settings saved in the state field, you must also insert the TVIF_STATE flag in mask. As described, the various TVIS flags that are to be defined must be set in stateMask. If the attribute in question is to be switched on, insert the corresponding constant in state at the same time. If the constant is not set in state, the attribute is deleted. For example, the following partial code changes the label of a given item.

```

TV_ITEM tvi;

tvi.hItem = ( the item to be changed )
tvi.mask = TVIF_TEXT;
tvi.pszText = "Hello World_";
TreeView_SetItem( hTreeView, &tvi );

```

Follow a similar procedure to find out an item's current settings. While you use a different message (TVM_GETITEM) and a different function macro (TreeView_GetItem), here, too, the address of a TV_ITEM structure is expected in the lParam. The handle of the TreeView item to be queried must be loaded in the hItem field. Moreover, prior to the call you must also insert the TVIF flags of the various fields, in which the TreeView data is to be returned, in the mask field.

```
#define TreeView_GetItem(hwnd, pItem) \
    (BOOL)SendMessage(hwnd, TVM_GETITEM, 0, (LPARAM)(TV_ITEM FAR*)(pItem))
```

If you specify TVIF_TEXT in mask, you have to initialize the TV_ITEM.pszText field with the address of the buffer you just allocated. Indicate the size of the address in TV_ITEM.cchMaxText. By specifying TVIF_STATE in mask, you get back the current attributes of the item in the state field. Test the TVIS flags to determine which attribute is active. For example, if the TVIS_SELECTED attribute is set, you have the currently selected item. The following example shows how to use this flag to check whether a given item is selected.

```
TV_ITEM tvi;

tvi.mask = TVIF_STATE;                                // Query state
tvi.hItem = ( of the item to be checked )             // of which entry?
if( TreeView_GetItem( hTreeView, &tvi ) )
    if( tvi.state & TVIS_SELECTED )
    {
        ...                                           // Item is selected
    }
```

Setting the graphics

One outstanding feature of the TreeView is its capacity to display a graphic next to each item, which can be set by the caller as desired. We've mentioned two fields, TV_ITEM.iImage and TV_ITEM.iSelectedImage, in conjunction with the insertion of new items. You use them when you link the TreeView to an Image List. From this point on, the TreeView will take the graphics from this Image List. The size of graphics is not preset. Instead, it's determined by the Image List.

Also, each item can include a state icon that will appear next to the actual icon. It's intended to describe the current state of the item in greater detail. You can use any Image List you want for the state images, you don't have to get them from the same image list as the actual icons. That's why the state images can be larger or smaller than the icons.

The TreeView provides the TVM_SETIMAGELIST message for setting one of the two Image Lists. You can also use the TreeView_SetImageList() macro to send this message. With this message, the TreeView expects the handle of the Image List in the wParam, while it expects either the TVSIL_NORMAL (for the image list of the standard icons) or TVSIL_STATE (for the image list with the state images) constant in iImage.

```
#define TreeView_SetImageList(hwnd, himl, iImage) \
    (HIMAGELIST)SendMessage(hwnd, TVM_SETIMAGELIST, iImage, \
    (LPARAM)(UINT)(HIMAGELIST)(himl))
```

You can also use the same message to detach an image list from a TreeView, which will then be forced to do without its images. To do that, simply specify NULL in himl and one of the two TVSIL constants in ImageList. The graphics next to the individual items will disappear instantly.

To determine the handle of the currently set Image List, use the TVM_GETIMAGELIST message or the corresponding function macro TreeView_GetImageList(). The only argument that is expected is one of the two TVSIL constants to specify the Image List to be queried.

```
#define TreeView_GetImageList(hwnd, iImage) \
    (HIMAGELIST)SendMessage(hwnd, TVM_GETIMAGELIST, iImage, 0)
```

As a function result, you are returned the handle of the image list set or NULL if either the specified TreeView handle was wrong or no image list was set in the TreeView.

Overlays and state images

Overlays and state images are two other images that provide support to an item's actual icon. You can define an overlay image for each item that can be inserted over the image. To use an overlay, you must first define it within the context of the respective Image List (overlay images are taken from the same image list that contains the icons of the items). Up to four overlay images can be defined within an Image List, numbered 1 through 4. The ImageList_SetOverlayImage() function performs this task.

One of the four overlays can be inserted over the icon of an item. Use the state and stateMask field of a TV_ITEM item to determine which overlay and whether you're even able use this option. When inserting (TVM_INSERTITEM) or changing an item (TVM_SETITEM), insert the TVIS_OVERLAYMASK constant in the stateMask field. Using its number, specify the desired overlay image in the state field. Add the predefined macro, INDEXTOOVERLAYMASK(), which is fed with the number between 1 and 4 of the overlay image, to the desired TVIS flag. The following code sequence makes this clear.

```
TV_ITEM tvi;

tvi.mask = TVIF_STATE;
tvi.hItem = ( Item to be changed );
tvi.stateMask = TVIS_OVERLAYMASK;           // Overlaymask is valid
tvi.state = INDEXTOOVERLAYMASK(1);          // First overlay image
TreeView_SetItem( hTreeView, &tvi );        // Set item
...
...
tvi.mask = TVIF_STATE;
tvi.hItem = ( Item to be changed );
tvi.stateMask = TVIS_OVERLAYMASK;           // Overlay mask is valid
tvi.state = INDEXTOOVERLAYMASK(0);          // No overlay image
TreeView_SetItem( hTreeView, &tvi );        // Set item
```

Setting the state image

In addition to the overlay, each item can also have a “state image”, describing its current status. A “state image” is, for example, a check mark that indicates whether the item is currently active. The state images are not taken from the same Image List as the actual icons (at whose left side they appear). Instead, they come from a separate Image List which can be set using TVM_SETIMAGELIST. To display a state image, set the TVIS_STATEIMAGEMASK flag in stateMask and add the predefined macro INDEXTOSTATEIMAGEMASK() to the desired flags in state. Pass the desired state image to the macro in the form of the index number of the image from the Image List set for state images. The following code sequence shows how such an image is inserted and later removed.

```
TV_ITEM tvi;

tvi.mask = TVIF_STATE;
tvi.hItem = ( Item to be changed );
tvi.stateMask = TVIS_STATEIMAGEMASK;        // State image is valid
tvi.state = INDEXTOSTATEIMAGE(5);           // Number of the state image
TreeView_SetItem( hTreeView, &tvi );        // Set image
...
...
tvi.mask = TVIF_STATE;
tvi.hItem = ( Item to be change );
tvi.stateMask = TVIS_STATEIMAGEMASK;        // State image-mask is valid
tvi.state = INDEXTOSTATEIMAGE(0);           // No state icon
TreeView_SetItem( hTreeView, &tvi );        // Set image
```

Switching between display modes

When creating or changing an item, you always specify only the indexes of the two images for selected and unselected status. However, what if a parent is opened? Do we need to display a different image in that case, which represents that item in the open state? To do that, you have to respond to the opening and closing of an item. The next section shows you how this works.

Opening and closing

The TreeView control does most of the work for the caller as far as the opening and closing of parent items and the redrawing of the tree it involves are concerned. Double-click on the label of an item or else click the plus/minus symbol (TVS_HASBUTTONS TreeView style) to have the TreeView automatically switch between opened and closed.

However, the TVN_ITEMEXPANDING notification is used first to give the parent window the option of opposing the status change. This gives the application the opportunity to keep a user away from certain areas. As with the other notifications, you receive a pointer to a structure of the NM_TREEVIEW type in lParam of the message (see deletion of items).

```
typedef struct {                                // Describes TreeView item during notification
    NMHDR      hdr;                               // The usual notification header
    UINT       action;                             // One of the TVE constants
    TV_ITEM    itemOld;                            // No significance
    TV_ITEM    itemNew;                           // TV_ITEM structure with item in question
    POINT      ptDrag;                             // No significance
} NM_TREEVIEW, FAR *LPNM_TREEVIEW;
```

You will find various data on the item in the itemNew field of the TV_ITEM structure. This includes the TreeItem handle (hItem), the current status (state) and the lParam. The NM_TREEVIEW.action field determines what is to happen to the item. One of the following TVE constants is specified in this field:

TVE constants (TreeView Expand) describe the opened state of a parent item	
Constant	Meaning
TVE_COLLAPSE	Closes the parent item and conceals the child items.
TVE_EXPAND	Opens the parent item and makes the child items visible.
TVE_TOGGLE	Switches the parent between open and closed.
TVE_COLLAPSERESET	Closes parent item and deletes all child items at the same time.

The parent window decides, based on its return value, whether the desired action may be executed. The action is confirmed if FALSE is returned. TRUE, however means a strict “no”. If the action is allowed, you then also receive a TVN_ITEMEXPANDED message automatically with the same information as in the preceding TVN_ITEMEXPANDING message. This message gives the caller the chance to keep the status of certain items intact.

The two notifications are also where the fields iImage and iImageSelected in the TV_ITEM structure can be switched if the item is to get a new icon. To do this, simply use the TVM_SETITEM message (preferably right after receiving TVN_ITEMEXPANDING). The item is already switched and can be redrawn in the case of TVN_ITEMEXPANDED so the late switch requires an unnecessary screen operation.

Programmed change

The TreeView opens and closes parent items, but not only as a response to an action by the user. The software, too, can implement opening and closing by using the TVM_EXPAND message. The corresponding macro is called TreeView_Expand().

```
#define TreeView_Expand(hwnd, hitem, code) \
    (BOOL)SendMessage((hwnd), TVM_EXPAND, (WPARAM)code, \
        (LPARAM)(HTREEITEM)(hitem))
```

The macro shows that a code is expected in the wParam. This code simply represents one of the above TVE constants and tells the TreeView what to do. The TreeItem handle of the item is also expected in lParam. The Boolean function value tells you whether it was possible to execute the action; possibly the parent window has also refused. The following code shows how the display of the currently selected item is switched between opened and closed.

```
HTREEITEM hItem;
TV_ITEM tvi;

// Get currently selected item _____
hItem = TreeView_GetSelection( hTreeView );

if( hItem ) // Is there a selected item? _____
{
    // The following code sequence forms the call to _____
    // TreeView_Expand( hTreeView, hItem, TVE_TOGGLE ); after _____
    tvi.hItem = hItem;
    tvi.mask = TVIF_STATE;           // Get status of the item
    TreeView_GetItem( hTreeView, &tvi );
    if( tvi.state & TVIS_EXPANDED )
        TreeView_Expand( hTreeView, hItem, TVE_COLLAPSE );
    else
        TreeView_Expand( hTreeView, hItem, TVE_EXPAND );
}
```

Indentation of the child items

A TreeView control manages the width of the indentation for all items as a unit. The indentation indicates the hierarchical arrangement of parent items and their child items. It's set in pixels and can be queried with the TVM_GETINDENT message and set using the TVM_SETINDENT message. Accordingly, two macros with the names TreeView_GetIndent() and TreeView_SetIndent() are available.

```
#define TreeView_GetIndent(hwnd) \
    (UINT)SendMessage((hwnd), TVM_GETINDENT, 0, 0)

#define TreeView_SetIndent(hwnd, indent) \
    (BOOL)SendMessage((hwnd), TVM_SETINDENT, (WPARAM)indent, 0)
```

The macros show the following: In both cases, specify 0 for the lParam of the message. TVM_SETINDENT expects the new indentation in pixels instead. If it's below the lower limit set throughout the system, the macro uses the lower limit as a default.

Browsing a TreeView

Sometimes you'll want to browse a TreeView to determine its items or to look for a specific one. To do this, use the TVM_GETNEXTITEM message. Specify one of ten TVGN constants as a flag in the wParam of the message. In lParam, depending on the TVGN constant, specify the TreeItem handle of an item that will serve as the starting point for the search. As the function result of SendMessage(), you get back the TreeItem handle of the item found, or NULL, if no corresponding item was found.

The individual constants can be broken down into several groups with variable functionality. TVGN_CARET and TVGN_DROPHILITE return an item of the kind set only once per TreeView control at any given point in time. TVGN_CARET returns the currently selected item, TVGN_DROPHILITE returns the item highlighted in the context of a Drag and Drop operation as the potential drop target. Therefore, in both cases, the value the user specifies in lParam when the TVM_GETNEXTITEM is sent is of no importance.

TVGN constants (TreeView Get Next) determine the type of item searched		
Constant	Returns the...	hItem in IParam
TVGN_CARET	Selected item	Unimportant
TVGN_DROPHILITE	Item labelled as the potential target of a drag and drop operation	Unimportant
TVGN_PARENT	Parent item of the specified	Child item whose parent is being sought
TVGN_CHILD	First child item	Parent item whose child item is being sought.
TVGN_PREVIOUS	Preceding item on the same level	Item whose predecessor is being sought
TVGN_NEXT	Following item on the same level	The item whose successor is being sought
TVGN_FIRSTVISIBLE	First visible item in the client area of the TreeView	Unimportant
TVGN_NEXTVISIBLE	Next visible item in the client area (next line)	A visible item preceding the one being sought
TVGN_PREVIOUSVISIBLE	Preceding item visible in the client area (preceding line)	A visible item following the one being sought
TVGN_ROOT	Uppermost item in the root of the TreeView	Unimportant

Use TVGN_PARENT and TVGN_CHILD to jump from branch to branch in the hierarchy. TVGN_CHILD returns the first subitem to an item entered in the IParam, if one exists. That means you are jumping one level lower in the tree. Use TVGN_PARENT to jump in the opposite direction, from an item to its parent item, moving one branch forward toward the root of the tree. To start at the root of the tree, use TVGN_ROOT to get the TreeItem handle of the item at the root of the tree.

Once you are in a branch, you may want to browse through the items within this branch. Use TVGN_PREVIOUS and TVGN_NEXT for that purpose. The first constant returns the predecessor of an item entered in the IParam, while the second constant returns its successor.

While the constants we've mentioned up to now move you through the entire tree, TVGN_FIRSTVISIBLE, TVGN_NEXTVISIBLE and TVGN_PREVIOUSVISIBLE have to do exclusively with the items currently visible in the client area of the control. TVGN_FIRSTVISIBLE takes you to the TreeItem handle of the topmost item visible in the client area. Use this in a subsequent call to TVGN_NEXTVISIBLE to get the handle of the next visible item, that is, the item in the next screen line. It doesn't matter whether this item is located in the same branch or a different one. The same holds true for TVGN_PREVIOUSVISIBLE, which returns the item in the preceding screen line.

Separate macros are available for all these constants. The names and definitions of these macros reveal their purpose:

```
#define TreeView_GetChild(hwnd, hitem) \
    TreeView_GetNextItem(hwnd, hitem, TVGN_CHILD)

#define TreeView_GetNextSibling(hwnd, hitem) \
    TreeView_GetNextItem(hwnd, hitem, TVGN_NEXT)

#define TreeView_GetPrevSibling(hwnd, hitem) \
    TreeView_GetNextItem(hwnd, hitem, TVGN_PREVIOUS)

#define TreeView_GetParent(hwnd, hitem) \
    TreeView_GetNextItem(hwnd, hitem, TVGN_PARENT)

#define TreeView_GetFirstVisible(hwnd) \
    TreeView_GetNextItem(hwnd, NULL, TVGN_FIRSTVISIBLE)

#define TreeView_GetNextVisible(hwnd, hitem) \
```

```

TreeView_GetNextItem(hwnd, hitem, TVGN_NEXTVISIBLE)

#define TreeView_GetPrevVisible(hwnd, hitem) \
    TreeView_GetNextItem(hwnd, hitem, TVGN_PREVIOUSVISIBLE)

#define TreeView_GetSelection(hwnd) \
    TreeView_GetNextItem(hwnd, NULL, TVGN_CARET)

#define TreeView_GetDropHighlight(hwnd) \
    TreeView_GetNextItem(hwnd, NULL, TVGN_DROPHILITE)

```

This program code uses the `TreeView_GetNextItem()` and `TreeView_GetPrevVisible()` functions to browse through a tree recursively with `TVM_GETNEXTITEM` and display the labels of all the items with `printf()`. The starting point is `TreeView_GetRoot()`. It receives the handle of the `TreeView` and uses `TreeView_GetNextItem()` to browse the tree starting from the root.

```

void TraverseTreeEngine( HWND      hTreeView,
                        HTREEITEM hParent,
                        int       iLevel )
{
    while( hParent ) // Run through all siblings
    {
        TV_ITEM tvi;
        char szText[ MAX_PATH ];

        // Query data of the item
        tvi.mask = TVIF_TEXT |
                  TVIF_IMAGE |
                  TVIF_SELECTEDIMAGE |
                  TVIF_PARAM |
                  TVIF_STATE |
                  TVIF_HANDLE;
        tvi.hItem = hParent;
        tvi.pszText = szText;
        tvi.cchTextMax = sizeof( szText );

        if( TreeView_GetItem( hTreeView, &tvi ) )
        {
            printf( "%s\n", szText );

            // Run through child items
            TraverseTreeEngine( hTreeView,
                               TreeView_GetChild( hTreeView, hParent ),
                               iLevel + 1 );
        }
        // Find next sibling
        hParent = TreeView_GetNextSibling( hTreeView, hParent );
    }
}

void TraverseTree( HWND hTreeView )
{
    TraverseTreeEngine( hTreeView, TreeView_GetRoot( hTreeView ), 0 );
}

```

Getting the number of items

If you want to browse through all the items in a `TreeView`, sometimes it's a good idea to determine the number of items in the control first. For example, you could do this when allocating an array on the heap that is to receive specific data for each item. It's easy to do this using the `TVM_GETCOUNT` message and the `TreeView_GetCount()` macro. You are returned the total number of items as the function result of the `SendMessage()` call.

```
#define TreeView_GetCount(hwnd) \
    (UINT)SendMessage((hwnd), TVM_GETCOUNT, 0, 0)
```

Selecting the current item

While `TreeView` controls and monitors the selection of items by the user, it gives an application the opportunity to track this process. Whenever a new item is to be selected, the control sends the parent window a `TVN_SELCHANGING` message which gives the owner the chance to confirm or reject the selection. So the software is free to suppress certain items, if this seems appropriate. The message procedure must then return `TRUE` as its response to the `TVN_SELCHANGING` message.

The `lParam` of the `WM_NOTIFY` message, under whose auspices the `TVN_SELCHANGING` message travels, points to a structure of the `NM_TREEVIEW` type. You'll find two `TV_ITEM` items here. They describe the item that is selected and the one to be selected.

```
typedef struct {                                // For TVN_SELCHANGING and TVN_SELCHANGED
    NMHDR      hdr;                             // The usual notification header
    UINT       action;                          // One of the TVC Constants
    TV_ITEM     itemOld;                        // The previously selected item
    TV_ITEM     itemNew;                       // The new item to be selected
    POINT       ptDrag;                         // No significance
} NM_TREEVIEW, FAR *LPNM_TREEVIEW;
```

The `hItem`, `state` and `lParam` fields are set in the two `TV_ITEM` variables, `itemOld` and `itemNew`, so the old and the new can be identified. The `NM_TREEVIEW.action` field tells you what produced the change. You will find one of three TVC constants there (see the following table):

If you allow the change, you will soon receive a `TVN_SELCHANGED` message with precisely the same parameters as the preceding `TVN_SELCHANGING`. That gives the owner the opportunity to take note of the new selection, if necessary.

TVC constants (TreeView Caret) return the reason for the change in selection	
Constant	Meaning
<code>TVC_BYKEYBOARD</code>	It was the keyboard!
<code>TVC_BYMOUSE</code>	No, the mouse!
<code>TVC_UNKNOWN</code>	I don't know.

Program-controlled changing of the selection

Also, an application can actively change the selection by sending the `TreeView` control a `TVM_SELECTITEM` message. On close examination, the message turns out to be a truly versatile talent, for you can also use it to perform two other operations. Use one of three constants in the `wParam` of the message to indicate the operation you want:

TVGN constants (TreeView Get Next) are also used in the <code>TVM_SELECTITEM</code> message	
Constant	Meaning
<code>TVGN_CARET</code>	The specified item is made the current selection.
<code>TVGN_FIRSTVISIBL</code>	The specified item is shown as the topmost visible item in the <code>TreeView</code> control.
<code>TVGN_DROPHILITE</code>	The specified item is displayed in such a way that it is recognizable as the potential target of a Drag and Drop operation.

Specify the `TreeItem` handle of the item in the `lParam` of the message. The function result of `SendMessage()` shows whether it was possible to execute the action as desired. In this case, it reads `TRUE`. The three possible operations of the `TVM_SELECTITEM` message are mapped to the following three macros in the `COMMCTRL.H` Include file:

```
#define TreeView_SelectItem(hwnd, hitem) \
    TreeView_Select(hwnd, hitem, TVGN_CARET)

#define TreeView_SelectDropTarget(hwnd, hitem) \
    TreeView_Select(hwnd, hitem, TVGN_DROPHILITE)

#define TreeView_SelectSetFirstVisible(hwnd, hitem) \
    TreeView_Select(hwnd, hitem, TVGN_FIRSTVISIBLE)
```

Displaying items

Under certain circumstances, you want to make sure that a specific item is visible on the screen, for example, because a search operation has returned precisely this item as the search result. The `TVM_ENSUREVISIBLE` message ensures that the item entered in `lParam` by means of its `TreeItem` handle is visible. In this case visible means the parent item above it is opened and the item appears in the viewing area of the `TreeView` control. It's not necessary to feed the `wParam`. The corresponding macro is called `TreeView_EnsureVisible()`.

```
#define TreeView_EnsureVisible(hwnd, hitem) \
    (BOOL)SendMessage((hwnd), TVM_ENSUREVISIBLE, \
        0, (LPARAM)(HTREEITEM)(hitem))
```

If parent items must be opened to display the item, the control sends `TVN_ITEMEXPANDING` messages to confirm its permission for opening them. The item can only be made visible when the control grants this permission, so that `TRUE` is obtained as the function result. If the result is `FALSE`, the item could not be displayed for some reason.

To determine how many items can be displayed at the same time within the client area of the `TreeView`, simply send the `TVM_GETVISIBLECOUNT` message to the `TreeView` control. It's not necessary to load `wParam` and `lParam`. The function result is the number of items.

```
#define TreeView_GetVisibleCount(hwnd) \
    (UINT)SendMessage((hwnd), TVM_GETVISIBLECOUNT, 0, 0)
```

Querying visibility

If you don't necessarily want to have an item displayed, but simply want to know whether it is visible, use the `TVM_GETITEMRECT` message. The message must be accompanied in `lParam` by the address of a `RECT` structure. The function places the coordinates of the item's bounding rectangle in this structure if the item is visible. The specified coordinates refer to the upper left corner of the control's client area.

Use the `wParam` to show whether the bounding rectangle is to be queried relative to the label of the item or the entire item with label and images. Specifying `TRUE` concerns only the label, whereas `FALSE` includes the entire item. You get a function result of `TRUE` if the item is currently visible. As an exception, the corresponding macro, `TreeView_GetItemRect()`, is broken down into two C-commands, separated by a comma here (a valid but seldom used form).

```
#define TreeView_GetItemRect(hwnd, hitem, prc, code) \
    (*(HTREEITEM FAR *)prc = (hitem), \
    (BOOL)SendMessage((hwnd), TVM_GETITEMRECT, \
        (WPARAM)(code), (LPARAM)(RECT FAR*)(prc)))
```

Assigning mouse events to items

To respond to an item that has been clicked, you are confronted with the problem of having to assign a mouse event to one of the two specified items of the `TreeView` controls. In this case, the `TVM_HITTEST` message will help you along. It determines the item at a given point and, in ideal circumstances, returns the `TreeItem` handle of the item. Additional information is in the `TV_HITTESTINFO` structure. You have to prepare this structure before sending the message and reference it using its `lParam`. Specify the point relative to the upper left corner of the control's client area in the `pt` field.

```
typedef struct _TV_HITTESTINFO {
    POINT      pt;                // Point to be loaded by the caller
    UINT       flags;             // Return: one of the TVHT constants
    HTREEITEM  hItem;             // Return: Handle of the item
} TV_HITTESTINFO, FAR *LPTV_HITTESTINFO;
```

In case of success, you get back the `TreeItem` handle of the item (identical to the function result) in the `hItem` field. One of the TVHT constants will indicate the detailed position of the specified point in the flags field:

TVHT constants (TreeView Hit Test) describe the item below the specified point	
Constant	Point lies below ...
Successful cases	
TVHT_ONITEMICON	The icon of the item
TVHT_ONITEMLABEL	The label of the item
TVHT_ONITEMINDENT	The indentation of the item
TVHT_ONITEMBUTTON	The plus/minus button of the item
TVHT_ONITEMRIGHT	To the right, next to the item
TVHT_ONITEMSTATEICON	The state icon of the item
TVHT_ONITEM	Combines the three flags TVHT_ONITEMICON, TVHT_ONITEMLABEL and TVHT_ONITEMSTATEICON.
Unsuccessful cases	
TVHT_ABOVE	Not below an item, because it's above the client area of the control
TVHT_BELOW	Ditto, below the client area
TVHT_TORIGHT	Ditto, to the right
TVHT_TOLEFT	Ditto, to the left

Use the `TreeView_HitTest()` function macro to save yourself the trouble of calling `SendMessage()`:

```
#define TreeView_HitTest(hwnd, lpht) \
    (HTREEITEM)SendMessage((hwnd), TVM_HITTEST, \
                           0, (LPARAM)(LPTV_HITTESTINFO)(lpht))
```

Processing a label

If you set the `TVS_EDITLABELS` flag in the window style of the `TreeView` control, the control offers the user the option of editing an item's label. However, before the control can insert an edit control above the item, allowing the user to edit, it makes sure the owner permits the editing. It sends a `TVN_BEGINLABELEDIT` message to the parent window for this purpose. It passes a pointer to a structure of the `TV_DISPINFO` type in the `lParam`. You can get all the necessary information on the item to be edited from this structure.

```
typedef struct {
    NMHDR    hdr;                // The usual header with the TVN_BEGINLABELEDIT code-
    TV_ITEM  item;               // Item concerned
} TV_DISPINFO;
```

The `hItem`, `state`, `pszText` fields and `lParam` are set in the `TV_ITEM` structure, so you can identify the item in question. To suppress editing, return a function result of `TRUE`, otherwise return `FALSE`, if the action can begin. This way you have the option of excluding individual items from the editing process, while leaving others open to it. After editing is complete, you get a `TVN_ENDLABELEDIT` message from the `TreeView`, signaling the end of the input. In turn, a `TV_DISPINFO` structure is returned in the `lParam`. You get the item from this structure's `TV_ITEM`. `hItem`, `state` and `lParam` are set, as before, and you

find out whether a new label was actually entered or input was cancelled from `pszText`. If `pszText` contains `NULL`, the user selected Cancel, in which case the item retains its old label. Otherwise, `pszText` points to a buffer with the new label of the item. The label is still not established in the item. The program has to send a `TVM_SETITEM` message with the new text to store the new label in the item permanently.

Programmed editing

However, an application can also require the user to supply the item with a new label, by having the edit control appear on its own. For this purpose, you need only send the `TreeView` a `TVM_EDITLABEL` message, or - alternatively - call the `TreeView_EditLabel()` function macro. The `TreeView` must already have the focus, otherwise it won't work.

```
#define TreeView_EditLabel(hwnd, hitem) \
    (HWND)SendMessage((hwnd), TVM_EDITLABEL, 0, (LPARAM)(HTREEITEM)(hitem))
```

Specify 0 in the `wParam` and the `TreeItem` handle of the item in `LPARAM` along with the `TVM_EDITLABEL` message. The item is selected automatically and made visible in the `TreeView` control, if not already visible. The program itself must allow the editing; or at the very least, not explicitly refuse it. For, as a result of the `TVM_EDITLABEL` message, the `TreeView` first sends off a `TVN_BEGINLABELEDIT` message, which the application could still use to stop editing.

Accessing the edit control

The application can even influence the input inside the edit control. It can do this by sending corresponding messages to the control or by putting it into a subclass to take over edit control. In both cases, possession of the edit control handle is required. Use `TVM_GETEDITCONTROL` to make this handle visible. Neither `wParam` nor `LPARAM` expects arguments in the call to `SendMessage()`. You receive the desired handle as the function result. You get a function result of `NULL` if the handle could not be supplied, for example, because no edit operation is active. As an alternative, you could also use the `TreeView_GetEditControl()` function macro.

```
#define TreeView_GetEditControl(hwnd) \
    (HWND)SendMessage((hwnd), TVM_GETEDITCONTROL, 0, 0)
```

If you manage input yourself, you have to signal the end of the input to the `TreeView` control using `TVM_ENDEDITLABELNOW`. If the input was not successfully completed, but rather cancelled, specify 0 in the `LPARAM` and `TRUE` in the `wParam`. In this case, the `TreeView` control does not read out the current contents of the edit control. It closes the edit control instead and retains the old label of the item.

```
#define TreeView_EndEditLabelNow(hwnd, fCancel) \
    (BOOL)SendMessage((hwnd), TVM_ENDEDITLABELNOW, (WPARAM)fCancel, 0)
```

Subclassing the edit control

Problems usually occur when working with the edit control if the `TreeView` control is part of a dialog box and the user presses ENTER or ESC during input to conclude the process. The corresponding `WM_KEYDOWN` messages are passed on immediately to the message procedure, which interprets them as a request for activation of the default buttons for these keys. (Usually OK in the case of `Enter` and Cancel for `Esc`.) As a result, the dialog box is closed although the user really just wanted to end editing of the label in the `ListView` control.

Avoid that by using the edit control to execute subclassing. Then you'll receive the messages addressed to the control before the control itself is able to respond to them. There is only one message you want to intercept in this way and handle differently than the edit control normally provides for: the `WM_GETDLGCODE` message. We'll talk about this in more detail later.

The following program code shows a section from a message procedure of the dialog box from our sample program, `TVDEMO`. You'll see how the program reacts when it receives a `TVN_BEGINLABELEDIT` message. This message marks the beginning of the editing of an item label. By sending the `TVM_GETEDITCONTROL` message using the `TreeView_GetEditControl()` macro, we get the handle of the edit control first. Then the program calls the API function, `SetWindowLong()`. This time we're calling it to change the address of the message procedure of the control and not to change

the window style of the control. Specify the `GWL_WNDPROC` constant to gain access to this parameter in the window structure of the control and are able to replace it. In this case, replace it with the address of the `EditSubclass()` function.

```

BOOL WINAPI DlgProc( HWND hWnd, UINT wMsg, WPARAM wp, LPARAM lp )
{
    switch( wMsg )
    {
        case WM_NOTIFY:
            switch( LOWORD( wp ) )
            {
                case IDC_TREEVIEW:
                {
                    LPNM_TREEVIEW pnmTV = (LPNM_TREEVIEW)lp;
                    switch( pnmTV->hdr.code )
                    {
                        case TVN_BEGINLABELEDIT:                // Input of new item label
                        {
                            HWND hEdit;

                            // Get handle of edit control...
                            hEdit = TreeView_GetEditControl( hTreeView );

                            // ...and prepare for receipt of all key presses —
                            g_lpOldEditProc = (FARPROC)SetWindowLong( hEdit,
                                                                    GWL_WNDPROC,
                                                                    (LONG)EditSubclass );
                        }
                    }
                    return FALSE;                                // Permit editing
                }
            }
            break;
        }
    }
    break;
}
return FALSE;
}

```

Instead of going to the default message procedure, which receives the edit control on the basis of its window class, all messages now go to the internal program function `EditSubclass()`. This can be seen in the following partial program listing. The function only responds to the one message, `WM_GETDLGCODE`. It passes all the other messages unchanged to the original message procedure of the control, whose address you noted in the global variable, `g_lpOldEditProc`, when setting the new procedure using `SetWindowLong()`. This way our message procedure doesn't have to take on any more work than absolutely necessary.

```

/*****
/* EditSubclass - Subclass function for EditControl of a TreeView */
/*-----*/
/* Parameter : default parameters */
/* Return value: default return value */
/*-----*/
/* Info: The EditControl requires all the keyboard input, since ESC */
/* and RETURN will otherwise be interpreted as cancelling */
/* the dialog box or pressing the default button. */
/*****

```

```

LRESULT WINAPI EditSubclass( HWND hWnd,
                             UINT wParam,
                             WPARAM wp,
                             LPARAM lp )
{
    // I want all keystrokes _____
    if( wParam == WM_GETDLGCODE ) return DLGC_WANTALLKEYS;

    // Call original EditBox function _____
    return CallWindowProc( g_lpOldEditProc, hWnd, wParam, wp, lp );
}

```

WM_GETDLGCODE is a message that a control receives from its dialog box when the dialog box has to determine which keys the control will respond to and which keys are the responsibility of the dialog box. An edit control normally responds by returning a function value. This value corresponds to the combination of the predefined constants DLGC_WANTARROWS and DLGC_WANTCHARS. As a result, the control captures the keyboard messages about the pressing of the arrow keys and normal letters and characters. However, this is not the case with the messages for **Enter** and **Esc**. These two messages go to the dialog box and trigger behavior which is not wanted here at all. As the above code shows, a different constant is returned when WM_GETDLGCODE is received, namely DLGC_WANTALLKEYS. This constant makes certain the control receives all of the keys while it has the focus. Therefore the control receives not only letters and arrow keys, but also **Enter** and **Esc**. That prevents the dialog box from responding to them.

You'll usually need to remove subclassing when you no longer need it, i.e., calling SetWindowLong() again to set the original message procedure. However, we don't need to worry about this in our example because the TreeView automatically deletes the edit control after the input is finished.

Sorting

You can use two TVM messages to a TreeView control to put the items within the tree into an orderly sequence. TVM_SORTCHILDREN executes an alphanumeric sort. Also, TVM_SORTCHILDRENCB involves the application in the sort by comparing two items using a callback function set by the application. That lets you change the alphanumeric sorting sequence (for example, by sorting in descending order) or use entirely different sorting criteria. These sorting criteria can be based on criteria that differ greatly from the item label (for example, entered in lParam of a TV_ITEM structure). First, here's the TreeView_SortChildren() function macro. It's used for automatic sorting with TVM_SORTCHILDREN:

```

#define TreeView_SortChildren(hwnd, hitem, recurse) \
    (BOOL)SendMessage( (hwnd), TVM_SORTCHILDREN, \
                        (LPARAM)hitem, (LPARAM)(HTREEITEM)hitem )

```

A TreeItem handle is passed to the message in the lParam, marking the starting point of the sort. It has to be an item with child items. Sending the message causes the child items to be sorted. Use the recurse parameter to specify whether the sort includes subordinate parent items and their children. TRUE stands for recursion.

Sorting places the items into a new order. However, this doesn't change their handles. In the expanded version, you have to give the message the address of a structure of the TV_SORTCB type, in the lParam. Fill the address of the structure beforehand. Pass the recursion flag in the wParam as before.

```

#define TreeView_SortChildrenCB(hwnd, psort, recurse) \
    (BOOL)SendMessage( (hwnd), TVM_SORTCHILDRENCB, \
                        (LPARAM)psort, (LPARAM)(LPTV_SORTCB)psort )

```

The control now finds out the starting point of the sort from the TV_SORTCB structure. The caller has to place it in the hParent field. The address of the callback function follows in lpfnCompare. lParam contains a value that must be specified with each call to the callback function so a link can be established between the callback and the initiator of the sorting operation.


```
typedef struct {
    HTREEITEM    hParent;           // TreeItem handle of the starting point
    PFNTVCOMPARE lpfnCompare;      // Address of the callback function for the
                                   // sort.
    LPARAM       lParam;           // Long value for the callback function
} TV_SORTCB, FAR *LPTV_SORTCB;
```

The declaration of the PFNTVCOMPARE function pointer type clarifies the syntax of the callback function.

```
typedef int (CALLBACK *PFNTVCOMPARE)(
    LPARAM lParam1,                // lParam from TV_ITEM of an item to be compared
    LPARAM lParam2,                // lParam from TV_ITEM of the other item
    LPARAM lParamSort              // The lParam of TVM_SORTCHILDRENCB
);
```

The first two parameters of the lParam type represent the two items to be compared. Instead of being represented by their TreeItem handle, the items are represented by the value from the lParam field. This is the value you specified when you created the item in the TV_ITEM structure. (This assumes the information needed for sorting is accessible through this parameter.) It doesn't matter whether the information is in the parameter or lParam represents a pointer to a data structure from which the required data is obtained. For example, you could use the lParam to store your own sorting order with values 1 to n. By comparing the two values, the callback has an easy time deciding which one to sort first. The one with the smaller number in lParam always goes first.

Regardless of how the callback makes its decision about the desired sorting order, it has to display the desired arrangement to its caller using the function result. What you see is listed in the table to the right.

A value less than 0 ...	if the item called first is to precede the one called second.
The value 0 ...	if the two items are identical.
A value greater than 0 ...	if the item called second is to precede that called first.

The following partial program listing shows how to use TVM_SORTCHILDRENCB to execute an alphanumeric sort in reverse order. You will need the label of the item. Use lParam to access this label. For that reason, the program initialized the lParam with the handle of the item after creation of an item by a subsequent call to TVM_SETITEM. By specifying lParam as the TreeItem handle, you can get and compare the text of the two given items within the sort function using TVM_GETITEM.

```
TV_SORTCB tvs;

tvs.hParent = hItem;                // Item whose child items are to be sorted
tvs.lpfnCompare = CompareFunc;      // Comparison function
tvs.lParam = (LPARAM)hTreeView;     // Handle of the TreeView as user parameter

TreeView_SortChildrenCB( hTreeView, &tvs, FALSE );           // Sort

....

// The lParam of each TreeItem was loaded beforehand using TreeView_SetItem()
// with the handle of the respective item!!!

int CALLBACK CompareFunc(LPARAM lParam1,                    // Handle of the first item
                        LPARAM lParam2,                    // Handle of the second item
                        LPARAM lParamSort)                  // Handle of the TreeView
{
    TV_ITEM tvi1, tvi2;
    char Text1[ MAX_PATH ],
```

```

    Text2[ MAX_PATH ];

    tvil.mask      = TVIF_TEXT;
    tvil.pszText    = Text1;
    tvil.cchTextMax = sizeof( Text1 );
    tvil.hItem      = (HTREEITEM)lParam1;
    TreeView_GetItem( (HWND)lParamSort, &tvil );    // Label of the first item

    tvi2.mask      = TVIF_TEXT;
    tvi2.pszText    = Text2;
    tvi2.cchTextMax = sizeof( Text2 );
    tvi2.hItem      = (HTREEITEM)lParam2;
    TreeView_GetItem( (HWND)lParamSort, &tvi2 );    // Label of the second item

    return lstrcmp( Text1, Text2 );                // Compare
}

```

Owner-draw

The TreeView control does not recognize owner-draw in the true sense. Owner-draw lets an application draw the items of a TreeView on its own. However, an application can influence an item's label during display. That makes it possible for the application to customize the label each time it draws it. This procedure makes sense, not only if you want to influence the displayed texts dynamically, but also if the individual labels are to be maintained within their own data structures. So why place them in the TreeView if they are already available in memory?

The TreeView control supports this procedure with a notification message called TVN_GETDISPINFO. It sends this message to the parent window whenever it needs to get an item's label during the display. This is not the normal procedure. Usually you specify the item's text in the TV_ITEM.pszText field when creating the item using the TV_INSERTITEM message, so there is no need to get the text once more during the display. Instead of a concrete string pointer, an application can also specify the predefined constant LPSTR_TEXTCALLBACK for TV_ITEM.pszText when creating an item. This ensures the text will be queried during display each time using the TVN_GETDISPINFO message.

So an application does not need to determine whether any labels must first be queried during the display. Instead, an application can decide for each item on an individual basis how it wants to make the text available. When a TVN_GETDISPINFO message arrives, the application receives a pointer to a TV_DISPINFO structure in the lParam of the message. You will find a TV_ITEM structure in the item field, in which the hItem, state and lParam fields describe the item. Only pszText is not yet initialized. pszText has to be initialized now so the control can display the text.

```

typedef struct {
    NMHDR    hdr;                // The usual notification header
    TV_ITEM  item;               // Item concerned
} TV_DISPINFO;

```

***2**

Naturally, message processing needs to be concluded quickly. Also, this is not the time to worry about cleaning up the hard drive. Unnecessary dawdling causes considerable loss of speed during display. Taking advantage of this option means using the TVN_SETDISPINFO notification. This occurs when the user has edited an item's text whose label was set with LPSTR_CALLBACK. Because the text is not stored in the item itself, the owner must be able by using this message to accept the change into his own text memory. If this doesn't work, the owner will return the old text again with the next TVN_GETDISPINFO message and leave the user wondering what happened to his entry.

In the case of TVN_SETDISPINFO, you receive the address of a TV_DISPINFO structure with the item in the item field in the lParam again. The hItem, state and lParam fields of the TV_ITEM structure are initialized so you can identify the text. A pointer to a string with the new text of the item entered by the user is contained in pszText.

Responding to mouse and keyboard events

Although the control recognizes and controls the opening and closing of items on its own, under certain circumstances you'll also want to give your own response to mouse and keyboard events in connection with a TreeView control. For example, you could trigger a fixed action after the clicking of an item or a double-click. To accomplish this, you need to send messages. Just as it did with TreeView notifications (the TVN messages), the control uses a WM_NOTIFY message for this end, sending it to the parent of the control. As usual, the lParam refers to an NMHDR structure. You can determine the sender of the message (hwndFrom field) and the actual message code (code field) from this structure.

```
typedef struct {
    HWND  hwndFrom;           // Window handle of the sender
    UINT  idFrom;             // Control-ID of the sender
    UINT  code;               // The message code
} NMHDR;
```

Because this procedure is standardized for an entire series of controls, you don't get back any special TVN messages in the case of mouse events. Instead you get general NM messages. As the following table shows, these messages even show the parent the transmission and the loss of the focus.

NM messages (Notification Messages) describe interactive events.	
Message	Event
NM_CLICK	A mouse click above the client area with the primary (left) mouse button.
NM_DBLCLICK	A double-click above the client area with the primary (left) mouse button.
NM_RCLICK	A mouse click above the client area with the secondary (right) mouse button.
NM_RDBLCLICK	A double-click above the client area with the secondary (right) mouse button.
NM_RETURN	The user pressed the Enter key while the control had the focus.
NM_SETFOCUS	The control received the focus.
NM_KILLFOCUS	The control lost the focus.

At any rate, the information in the NMHDR structure is somewhat scanty. More information is preferable, particularly with mouse events (specifically the mouse position and the handle of the item lying under it). However, you can also obtain this data yourself. First, use the API GetCursorPos() function to load the mouse position into a POINT structure specified during the call. This will return the screen coordinates, which must first be set in relation to the client area of the control. Next, call the API ScreenToClient() function with the returned POINT structure and the handle of the TreeView control. That recalculates the coordinates relative to the upper left corner of the client area of the control. The declaration of both API functions is found in the Include file, WINUSER.H.

After the conversion you want to determine to which item the mouse event refers. Use the TVM_HITTEST message for that purpose. You can also send it using the corresponding macro, ListView_HitTest(). If a value not equal to NULL is received after sending the message with SendMessage(), the indicated coordinates could not actually be assigned to the item in the control. The return value corresponds in this case to the TreeItem handle of the item. It's also recorded in the TV_HITTESTINFO structure and in the hItem field. The following program code shows how the parent of a TreeView control responds in its message procedure to the receipt of an NM_DBLCLICK message and then determines the addressed item and its label. The latter is then displayed with MessageBox().

```
/* ***** */
/* DlgProc - Dialog function */
/* _____ */
/* Parameter : standard parameter */
/* Return value: standard return value */
/* ***** */
BOOL WINAPI DlgProc( HWND hWnd, UINT wParam, WPARAM wp, LPARAM lp )
```

```

{
    // Static variables for simplified access  _____
    static HWND      hTreeView;                // Handle of the TreeView

    switch( wParam )
    {
        case WM_NOTIFY:
            switch( LOWORD( wParam ) )
            {
                case IDC_TREEVIEW:
                {
                    LPNM_TREEVIEW pnmTV = (LPNM_TREEVIEW)lparam;
                    switch( pnmTV->hdr.code )
                    {
                        case NM_DBLCLK:                // Double-click
                        {
                            TV_HITTESTINFO tvhst;
                            HTREEITEM hItem;

                            GetCursorPos( &tvhst.pt ); // Determine mouse coordinates
                            ScreenToClient( hTreeView, &tvhst.pt ); // in relationship
                                                                    // to TV coordinates

                            // Determine item under cursor _____
                            hItem = TreeView_HitTest( hTreeView, &tvhst );
                            if( hItem )                // Some item hit?
                            {
                                TV_ITEM tvi;
                                char szText[ MAX_PATH ];

                                tvi.mask = TVIF_TEXT;                // Fetch item text
                                tvi.hItem = hItem;
                                tvi.pszText = szText;
                                tvi.cchTextMax = sizeof( szText );
                                TreeView_GetItem( hTreeView, &tvi );

                                MessageBox( hWnd, szText, "Double-click", 0 );
                            }
                            else MessageBox( hWnd, "Next to it!", "Double-click", 0 );
                        }
                    }
                    break;
                }
            }
        }
    }
    break;
}
return FALSE;
}

```

Responding to keyboard events

Besides the NM_RETURN as the response to the pressing of the Enter key, the parent receives the code of the pressed key using a special TreeView notification called TVN_KEYDOWN. It receives the message in the context of a WM_NOTIFY

message. It appears when the ListView control has the focus and the user presses any key. In the lParam of the message you'll find the address of a TV_KEYDOWN structure, containing more detailed information on the event.

```
typedef struct {
    NMHDR    hdr;           // Usual NMHDR structure with information on the
                           // sender of the message
    WORD     wVKey;         // Virtual key-code of the pressed key
    UINT     flags;         // 0
} TV_KEYDOWN;
```

Along with the usual NMHDR structure, whose code field contains the code of TVN_KEYDOWN, the virtual key-code of the pressed key is specified in wVKey. This key is evaluated and responds according to the logic desired. The flags field is at present not used and is always initialized with 0.

Drag and Drop

The hierarchies in the TreeView do not need to remain static. Use a Drag and Drop implementation to allow the user to move items and entire branches at runtime.. This does require a lot of work because a simple function call or the sending of a TVM message unfortunately won't work. The message procedure of the parent window must include additional program code to respond to various messages. At first it may seem rather complicated. However, the appropriate code will become easier once you've become familiar it. This is also true for the processes which cause the involved messages and the data structures used to be transmitted more or less directly to other common controls (for example, to the TreeView control).

The following program code concerns the TVDEMO program as it appears at the end of this chapter. It allows the user a very flexible configuration of the tree by simple dragging and dropping of items or complete branches.

Start of a drag operation

The TreeView control discovers the beginning of a drag operation by the user on its own and reports it to the parent window of the control using a TVN_BEGINDRAG or TVN_BEGINRDRAG notification. The one indicates the beginning of a drag operation with the primary (left) mouse button (TVN_BEGINDRAG), whereas the other concerns a drag operation with the right mouse button (TVN_BEGINRDRAG).

Do not explicitly set the TVS_DISABLEDRAHDROP flag in the window style of the control. This prevents any drag operation at the outset by having the control simply not respond to it, and it won't send a TVN_BEGINDRAG notification to the parent window. However, you receive one of the two messages, you'll find a pointer to an NM_TREEVIEW structure in the lParam of the message.

```
typedef struct {
    NMHDR    hdr;           // The message header with the code of the notification
    UINT     action;         // No significance
    TV_ITEM  itemOld;        // Not initialized
    TV_ITEM  itemNew;        // TV_ITEM structure with new item
    POINT    ptDrag;         // Mouse coordinates for drag operation
} NM_TREEVIEW;
```

The message, travelling in the baggage of a WM_NOTIFY message, uses the hdr field to establish its identity as a TVN_BEGINDRAG message, and identifies the control to be sent. The TV_ITEM item in the itemOld field is not initialized, but the one in itemNew describes the item dragged by the user. The hItem, state and lParam fields in this structure are initialized to identify the item.

The ptDrag field in the NM_TREEVIEW structure is also initialized with the coordinate of the point on which the user clicked at the beginning of the drag operation. The point lies within the circle of influence of the item specified in itemNew. Otherwise, the control would not have triggered a drag operation for this item. It's measured relative to the upper left corner of the client area of the control and will soon be needed. You'll usually want to record the data of the item from itemNew in a static variable.

A few more messages will appear in the window procedure of the parent before you can use this information to respond to the dropping of the item by the user.

Setting the drag image

The next step is to first build an image which is dragged across the screen together with the mouse pointer in the context of a Drag and Drop operation. The TreeView control provides a special message called TVM_CREATEDRAGIMAGE for this purpose. You can also call this message with the corresponding macro TreeView_CreateDragImage.

```
#define TreeView_CreateDragImage(hwnd, hitem) \
    (HIMAGELIST)SendMessage(hwnd, TVM_CREATEDRAGIMAGE, \
        0, (LPARAM)(HTREEITEM)(hitem))
```

When the message is sent, the wParam contains 0 and the lParam contains the handle of the item, as you will find in NM_TREEVIEW.newItem.hItem. After sending the message, you receive the handle of an image list as the function result. The control generates this handle for the execution of the drag operation. The image list contains only a single image. This image includes the icon of the item and its label. It always trails the mouse pointer as “drag image” during the drag operation so the user can see the drag process.

Before starting, the drag image from the returned image list must first be set as the drag image using the Image List function ImageList_BeginDrag(). The function expects the handle of the image list, the index of the image (here always 0) as well as the distance between the hotspot of the cursor and the upper left corner of the image.

```
BOOL ImageList_BeginDrag(          // Starts dragging with an image from the IL
    HIMAGELIST himl,              // Handle of the image list
    int iTrack,                    // Number of the image
    int dxHotspot,                 // X-distance of the drag point from the upper
                                // left-hand image corner
    int dyHotspot                  // ditto Y (both entries in pixels)
);
```

dxHotspot and dyHotspot represent the distance between the point over which the hotspot of the mouse pointer was positioned at the beginning of the drag operation and the upper left corner of the item lying under it. This distance needs to remain constant while the drag image is moving across the screen. Otherwise, it seems to the user that the image is “floating” beneath the mouse pointer. However, we want it to move parallel to the mouse pointer. This is why this distance must be set when ImageList_BeginDrag() is called. Unfortunately, in the case of the TreeView control it’s virtually impossible to determine the distance between the mouse position at the drag point and the upper left corner of the item lying beneath it. However, this is precisely the information required here. That is why the Explorer also takes care of this by specifying 0 for both coordinates. We’ll use this procedure.

The current cursor position is determined in the next step with the API function GetCursorPos(). The returned coordinate is passed on to ImageList_DragEnter(), an Image List function which receives the handle of a window at the same time. It determines the window to which the drag operation will remain restricted. The drag image is trimmed at the borders of the specified window and so is no longer visible outside the window.

```
BOOL ImageList_DragEnter(
    HWND hwndLock, // Window to which the drag operation remains restricted
    int x,          // X-coordinate for start of display of the drag image in
                    // the window
    int y           // y-coordinate for start of display of the drag image in
                    // the window
);
```

Specifying NULL for the hwndLock parameter ensures that the entire Desktop will be considered the window for executing the drag operation. The specified coordinates refer to the upper left corner of the specified window and determine the position where the drag image appears for the first time. If the Desktop is selected, specify absolute screen coordinates.

Next, it's necessary to set the capture for mouse events to the parent window of the TreeView control (in this case the dialog box) using the API function `SetCapture()`. This ensures the user will be able to guide the mouse pointer over the screen without one of the controls that it "runs over" receiving any mouse messages. After all, the other controls on the screen should remain unused during the drag operation. After calling `SetCapture()`, all mouse messages go directly to the message procedure of the window specified during the call. This is true even if the mouse pointer is positioned outside its sphere of influence.

The following partial program code is an example of the steps mentioned in the operation. You'll see the beginning of the dialog box message procedure from the TVDEMO sample program at the end of the chapter. In this procedure, upon receipt of a `WM_NOTIFY` message, a response to the `TVN_BEGINDRAG` message occurs. Also, an internal drag flag called `bDragging` is set along with the actions mentioned. This flag is to remind the message procedure that it is still in a drag context the next time a message is received. Moreover, the handle of the dragged item is stored in the `hDragItem` variable. This lets you access it later when it is dropped somewhere.

```

BOOL WINAPI DlgProc( HWND hWnd, UINT wParam, WPARAM wp, LPARAM lp )
{
    // Static variables for simplified access
    static HWND      hTreeView;           // Handle of the TreeViews
    static BOOL      bDragging;           // Is an item being currently dragged?
    static HTREEITEM hDragItem;           // Item to be dragged
    static HTREEITEM hDropTarget;         // Item "hit"
    static HIMAGELIST hDragImage;         // Image for drag operation

    switch( wParam )
    {
        case WM_NOTIFY:
            switch( LOWORD( wParam ) )
            {
                case IDC_TREEVIEW:
                {
                    LPNM_TREEVIEW pnmTV = (LPNM_TREEVIEW)lp;
                    switch( pnmTV->hdr.code )
                    {
                        case TVN_BEGINDRAG:           // Begin drag operation
                        {
                            POINT pt;
                            // Determine drag image of the item to be dragged
                            hDragImage = TreeView_CreateDragImage( hTreeView,
                                                                    pnmTV->itemNew.hItem );

                            // Set drag-image hotspot and begin drag operation
                            if ( !ImageList_BeginDrag(hDragImage, 0, 0, 0 ) )
                                return FALSE;
                            // Hide own cursor
                            ImageList_SetDragCursorImage( g_hDragCursors,
                                                            0,
                                                            -CRSR_HOTX,
                                                            -CRSR_HOTY );

                            ShowCursor( FALSE );           // Switch off system cursor

                            pt.x = pnmTV->ptDrag.x;
                            pt.y = pnmTV->ptDrag.y;
                        }
                    }
                }
            }
    }
}

```

```

ClientToScreen( hTreeView, &pt );

// Begin drag operation and prepare buffer —
ImageList_DragEnter(NULL, pt.x, pt.y );

SetCapture( hWnd );                                // Capture mouse

bDragging = TRUE;                                // Drag operation stands at
hDropTarget = 0;                                // Still no DropTarget
hDragItem = pnmtv->itemNew.hItem;                // Mark drag item
    }
    break;
}
}
break;
}
break;

```

Trailing the drag image

Drag mode remains active after the start of the drag operation until the user releases the mouse button, causing the message procedure to receive a WM_LBUTTONDOWN or WM_RBUTTONDOWN message. The drag item must trail the mouse pointer with each mouse movement and accompanying receipt of a WM_MOUSEMOVE message. Also, you may want to validate the screen controls lying below the mouse pointer as potential drop targets (depending upon the current task). The TreeView control provides a special style flag for this purpose called TVIS_DROPHILITED. You can set this flag in the state field of the TV_ITEM structure using TVM_SETITEM.

This requires additional program code. This code must first identify the respective item below the drag image, to see whether it's suitable as the target (and thus the drop site) of the drop operation. The programming involved depends greatly on the job to be done. For example, do you only want to consider items inside the same TreeView control as drop targets or other screen controls as well? In this case, we want to keep with targets inside the current TreeView so the dragged item can be dropped only over another item. While one item is being dragged over another, the latter needs to be validated as a potential drop target using TVIS_DROPHILITED.

If you only want to have the drag image follow the mouse pointer, the drag operation is relatively easy at this point. The ImageList function ImageList_DragMove() has all the operations which must be executed to enable the drag image to wander over the screen. First, it makes the drag image transparent at its old position and then restores the old screen content before it draws the drag image at its new position. If the drag operation was started by calling ImageList_BeginDrag(), simply call ImageList_DragMove() each time a WM_MOUSEMOVE message is received, to trail the drag image. So, you need to enter only the two coordinates received in lParam in the context of the WM_MOUSEMOVE message as arguments.

You must first, of course, have recomputed the window entered in ImageList_DragEnter(), and that means in this case: the upper left corner of the screen. To do this easily, use the API ClientToScreen function. The following is the ImageList_DragMove() function. It receives the calculated coordinates as parameters.

```

BOOL ImageList_DragMove(
    int x,                                // New X-coordinate (relative to Desktop)
    int y                                // New Y-coordinate (relative to Desktop)
);

```

Now, watch out for the highlighting of the potential drag targets using TVIS_DROPHILITED style. Don't let it become a trap. Under certain circumstances, if the user moves the mouse pointer, it will leave the circle of influence of the highlighted item. Therefore, it must be redrawn in its original state. Furthermore, the mouse pointer will, under certain circumstances, be influenced by a different item. Then you have to highlight this new item as the drop target instead of the first.

It's easy to determine whether the mouse pointer is over a new item with the help of the TVM_HITTEST message. The only inconvenience is that the coordinate specified in the case of TVM_HITTEST must refer to the upper left corner of the control, while the coordinate you receive in the context of WM_MOUSEMOVE is oriented to the upper left corner of the window specified with SetCapture(). So, the code must perform a conversion. First, it calls the API function ClientToScreen() to map the coordinates of WM_MOUSEMOVE as absolute screen coordinates relative to the upper left corner of the screen. For this purpose, the function receives the window handle of the dialog box as well as the coordinates. Then call ScreenToClient(), specifying the handle of the TreeView along with the returned coordinates. The point determined from the conversion designates the distance of the mouse pointer from the upper left corner of the client area of the TreeView. That is the coordinate you require to send the TVM_HITTEST message. In this way you find out whether the mouse pointer and the drag image are now positioned over a potential drop target.

Highlighting the drop target

If you have to change the state of an item below the mouse pointer, and therefore its appearance, make certain the drag image is transparent beforehand. Otherwise, the original screen contents will be restored again at the original spot of the drag image the next time ImageList_DragMove() is called. However, the original screen contents no longer coincide with the display mode of the item, as it has just been set. Therefore, before changing the state of items beneath the drag image, call the ImageList function, ImageList_DragShowNoLock(), again to make the drag image transparent and permit changing (and thus drawing) the item below it at your leisure. Then call the function a second time to make the drag image visible again.

```

BOOL ImageList_DragShowNoLock(
    BOOL fShow                // Display drag image? TRUE or FALSE
);

```

The function expects the value TRUE as the only argument if the drag image is to appear and FALSE if it is to be hidden. This code segment clarifies this way of responding to a WM_MOUSEMOVE message during an active drag operation.

```

case WM_MOUSEMOVE:
{
    if( bDragging )                // Is an item currently being dragged?
    {
        HTREEITEM hItem;
        TV_HITTESTINFO tvhtst;

        // Determine mouse position _____
        tvhtst.pt.x = LOWORD( lp );
        tvhtst.pt.y = HIWORD( lp );

        // Determine item beneath the cursor _____
        hItem = TreeView_HitTest( hTreeView, &tvhtst );

        if( hItem != hDropTarget )
        {
            // Hide drag image _____
            ImageList_DragShowNoLock( FALSE );

            // Item under cursor is new DropTarget _____
            SetDropHighlight( hTreeView,
                             hItem,
                             hDropTarget );
            hDropTarget = hItem;                // Mark DropTarget

            // Make drag image visible again _____
            ImageList_DragShowNoLock( TRUE );
        }
    }
}

```

```

    }

    ClientToScreen( hTreeView, &tvhtst.pt);

    // Move drag image to mouse position
    ImageList_DragMove( tvhtst.pt.x,
                       tvhtst.pt.y );

    // Cursor shape indicates whether drag on DropTarget is allowed
    ImageList_SetDragCursorImage( g_hDragCursors,
                                  IsChildOf( hTreeView,
                                              hDragItem,
                                              hDropTarget )?1:0,
                                  -CRSR_HOTX,
                                  -CRSR_HOTY );
}
}
break;

```

Concluding the drag operation

The drag operation finally ends with the release of the mouse button. It's reflected within the message procedure as the receipt of a WM_LBUTTONDOWN (or WM_RBUTTONDOWN) message. The program knows the drop target over which the mouse pointer is positioned at this time from the WM_MOUSEMOVE received earlier. The dragged object must under certain circumstances be inserted at the drop target (depending upon the logic of the program). How that happens varies from case to case and according to the task being done.

The following partial code demonstrates the necessary steps for formally concluding the drag operation. It picks up where the two preceding partial code segments left off, and shows the end of the message procedure from our TVDEMO sample program. The program handles the WM_LBUTTONDOWN message, marking the end of the drag operation. First, ImageList_DragLeave() is called as a counterpart to the original ImageList_DragEnter(), which we used to declare the Desktop as the drag zone. Among other things, calling this function removes the drag image from the screen.

```

BOOL ImageList_DragLeave(
    HWND hwndLock    // Handle of the window in which no further dragging
                    // takes place
);

```

ImageList_EndDrag() is then called to end the drag operation officially. Parameters are not required.

```

void ImageList_EndDrag( );           // Ends current drag operation

```

Also, the Image List with the DragImage generated with TVM_CREATEDRAGIMAGE at the start of the drag operation is again released by calling ImageList_Destroy(). Finally, it's also necessary to release the capture again using the API function ReleaseCapture(). Then other windows will be able to receive mouse messages again.

Dragging to the drop point

The dragged item and all child items (if present) and the item at the drag point are dragged within the tree as a result of the drag operation. Different internal program functions, for example, CopyTree() or IsChildOf(), are called for that purpose. These appear in the listing of the TVDEMO sample program at the end of the chapter. IsChildOf() is used to initially determine whether the drop target is perhaps a child item of the drag item. In this case, the drag cannot be executed because that contradicts the logic of such an operation.

```

case WM_LBUTTONDOWN:
    if( bDragging )           // End drag operation?

```

```

{
    ImageList_EndDrag();           // Release buffer internal to ImageList
    ImageList_DragLeave(NULL);      // Remove DragImage from window

    ImageList_Destroy( hDragImage );           // Destroy drag image

    SetDropHighlight( hTreeView,           // Display DropTarget normally
        NULL,
        hDropTarget );
    if( hDropTarget )
    {
        // May partial tree be shifted to target?
        if( !IsChildOf( hTreeView, hDragItem, hDropTarget ) )
        {
            // Copy partial tree
            CopyTree( hTreeView, hDropTarget, hDragItem );
            // Delete "original"
            TreeView_DeleteItem( hTreeView, hDragItem );
        }
    }
    bDragging = FALSE;           // No further drag operation
    ReleaseCapture();            // Release capture
    ShowCursor( TRUE );
}
break;
}
return FALSE;
}

```

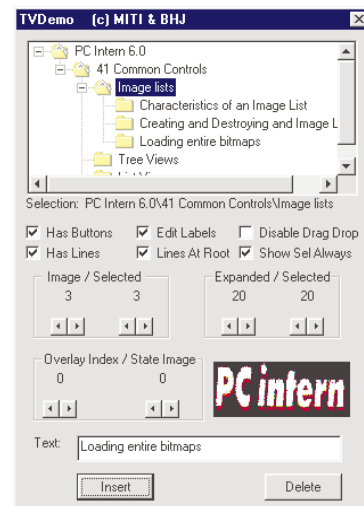
Sample program

Our sample program in this chapter is called TVDEMO and is a small TreeEditor. You will find the program in the \WIN95\TVDEMO directory of the PC INTERN CD. With the help of the program you can build a new tree by adding items and branches, moving them by Drag and Drop and assigning the desired text and an icon to them. At the end of the program, it's all stored in the current working directory in a file named TREEVIEW.DAT, and automatically loaded the next time you start the program.

Beneath the TreeView control you see several check boxes in the dialog box for the various window styles that you can assign a TreeView using the TVIS constants. These constants give you the option of experimenting with the various styles and familiarizing yourself with their effects. To edit an item's label in the TreeView control, you have to enable the Edit Labels check box.

Under the check boxes are small buttons for setting the icons of the item. They were taken from the System Image List. Their appearance depends on the current contents of the ImageList at program runtime. However, you determine which icon is used for which item. While only two images are stored in the TV_ITEM structure of an item with iImage and iSelectedImage, it is possible to select two pairs of the image and its selected image in TVDEMO. You can select one for

*TVDEMO.C
program is an
example of a
small TreeView
editor*

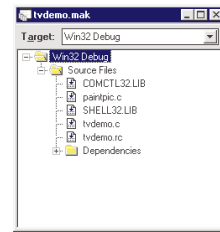


the item in closed state and one for the item in opened state. In this way, when opening and closing items the program can automatically switch between the set icons by setting the appropriate numbers in the `iImage` and `iSelectedImage` fields of the `TreeView` control.

The program stores the four “image information items” in the `IParam` of a `TV_ITEM` item. They fit wonderfully provided you use one byte for each. Although the number of possible images per setting is limited to 256, this is perfectly sufficient.

Also, an overlay image and a state image can be defined for each icon, these appear next to the icon of the item. The state images are also taken from the System Image List. Specify 0 for both images to keep them disabled. For example, in the figure above, the item with the text “6.2.1 ...” has a state image. The overlay images are automatically taken from the ones set in the System Image List. Use the `TVDEMO.MAK` make file to create the program. It's based on the `TVDEMO.C` file.

*TVDEMO.MAK
make file for the
TVDEMO
project*



You'll find the following program(s) on the companion CD-ROM



TVDEMO.C (C listing)

List Views

Besides the `TreeView` Control, the `ListView` Control is also included with the new generation of common controls. The `ListView` supports four display forms. You can easily switch between these during runtime. Windows 95 inserts this control at many points (in the Explorer, for example). The four possible views of the contents of a directory are the different display modes of a `ListView` control. These include large or small icons, as a list or as directories. However, you can't equate the `ListView` control with the Explorer. While the Explorer uses this control to display files, it must load their icons and names into the control itself. This is not a part of the control's functionality. The `ListView` control is solely responsible for the display of the entries passed to it. Where they come from and what they represent is unimportant. Even the dragging of files from of such a control using Drag and Drop is essentially an outgrowth of the functionality of the Explorer. The control makes the very basic functionality for this available but has no idea of what to do with documents, programs and shortcuts.

The `ListView` control is much more general and therefore suitable for more than simply displaying file contents. You can use this control whenever information is to be displayed to the user within a program in the form of directory lists or as a collection of graphics for selection.

Communication with ListViews

This section shows how to add `ListView` controls to your programs, how to load them with entries, display graphics with them, set columns, etc. On C level, the interface corresponds largely to that of the `TreeViews`. The declarations are found in the `COMMCTRL.H` file, the code of the control in the `COMCTL32.DLL`, and the window class is called “`SysListView32`”. This name is mapped to the `WC_LISTVIEW` constant within the Include file, which must be specified at runtime during the call to `CreateWindowEx()` to generate a `ListView` control as a window. For this to be successful requires a prior call to the `InitCommonControls()` function from the `COMCTL32.DLL`. Only then can you set up the `ListView` window class; otherwise, display of the control would be impossible.

Because the `ListView` control is presented in different views and likewise has additional features, the number of messages, constants and structures is even greater than that of the `TreeView` control. However, the semantics compared to `TreeView` and the other common controls have been largely retained. So, your work will be slightly easier if you have experience with `TreeView`. Many `ListView` reports and concepts will then for good reason seem very familiar. Only the prefixes are different. For example, you'll use `LVM_EDITLABEL` instead of `TVM_EDITLABEL` and the associated macro is called `ListView_EditLabel()` instead of `TreeView_EditLabel()`. The data structures are also similar. While in the `TreeView` you specified the address of a `TV_ITEM` structure in the `IParam` of a message, in the case of `ListView` use the `LV_ITEM` structure, which describes an item in the control. Before looking at this in detail, let's first look at the `LVM` messages used by an application to access a `ListView` control. They'll show what is possible with the `ListView` control.

An application manipulates a TreeView control by transmitting TVM messages.(Tree View Messages)		
LVM message	Associated macro	Task
LVM_ARRANGE	ListView_Arrange	Arranges the icons in the icon view.
LVM_CREATEDRAGIMAGE	ListView_CreateDragImage	Generates an image list with a drag image for an item.
LVM_DELETEALLITEMS	ListView_DeleteAllItems	Deletes all items of a ListView.
LVM_DELETECOLUMN	ListView_DeleteColumn	Deletes a column from the ListView.
LVM_DELETEITEM	ListView_DeleteItem	Deletes an item from the control.
LVM_EDITLABEL	ListView_EditLabel	Initiates the editing of the text of an item .
LVM_ENSUREVISIBLE	ListView_EnsureVisible	Ensures that a given item is visible in the client area of the control.
LVM_FINDITEM	ListView_FindItem	Finds a certain item in the ListView.
LVM_GETBKCOLOR	ListView_GetBkColor	Returns the background color of the control.
LVM_GETCOLUMN	ListView_GetColumn	Provides info on a given column of the ListView.
LVM_GETCOLUMNWIDTH	ListView_GetColumnWidth	Returns the width of a given column in pixels.
LVM_GETCOUNTPERPAGE	ListView_GetCountPerPage	Returns the number of lines which are completely visible in the client area at the same time.
LVM_GETEDITCONTROL	ListView_GetEditControl	Returns the handle of the edit control used for editing a text item.
LVM_GETIMAGELIST	ListView_GetImageList	Returns the handle of an image list set in the ListView.
LVM_GETISEARCHSTRING	ListView_GetISearchString	Returns the character string entered by the user for the incremental search.
LVM_GETITEM	ListView_GetItem	Returns various attributes of an item.
LVM_GETITEMCOUNT	ListView_GetItemCount	Returns the number of items contained in the control.
LVM_GETITEMPOSITION	ListView_GetItemPosition	Returns the position of an item in the two icon views.
LVM_GETITEMRECT	ListView_GetItemRect	Returns the screen region in which a given item is currently displayed in the two icon views.
LVM_GETITEMSPACING	ListView_GetItemSpacing	Returns the interval between the various items and elements of the ListView.
LVM_GETITEMSTATE	ListView_GetItemState	Returns the current status of an item .
LVM_GETITEMTEXT	ListView_GetItemText	Returns the text of an item.
LVM_GETNEXTITEM	ListView_GetNextItem	Returns the next item with a predetermined criterion.
LVM_GETVIEWORIGIN	ListView_GetViewOrigin	Returns the starting point of the viewing area which currently appears in the client area of the control.
LVM_GETSELECTEDCOUNT	ListView_GetSelectedCount	Returns the number of items currently selected.
LVM_GETSTRINGWIDTH	ListView_GetStringWidth	Returns the width of a string in pixels, measured on the font currently set in the control.
LVM_GETTEXTBKCOLOR	ListView_GetTextBkColor	Returns the background color for the texts displayed in the ListView.
LVM_GETTEXTCOLOR	ListView_GetTextColor	Returns the foreground color for the texts displayed in the ListView.
LVM_GETTOPINDEX	ListView_GetTopIndex	Returns the index of the uppermost item visible in the client area in the list and message display.
LVM_GETVIEWRECT	ListView_GetViewRect	Provides the bounding rectangle for all items visible in the client area.

An application manipulates a TreeView control by transmitting TVM messages.(Tree View Messages) - continued		
LVM message	Associated macro	Task
LVM_HITTEST	ListView_HitTest	Attempts to assign a screen position to an item lying below it.
LVM_INSERTCOLUMN	ListView_InsertColumn	Inserts an additional column in the ListView.
LVM_INSERTITEM	ListView_InsertItem	Inserts an additional item in the ListView.
LVM_REDRAWITEMS	ListView_RedrawItems	Tells the control to redraw a predetermined group of items.
LVM_SCROLL	ListView_Scroll	Scrolls the visible items of the ListView along the horizontal or vertical axis.
LVM_SETBKCOLOR	ListView_SetBkColor	Sets the background color of a control.
LVM_SETCOLUMN	ListView_SetColumn	Sets the attributes of a column.
LVM_SETCOLUMNWIDTH	ListView_SetColumnWidth	Sets the column width of a given column.
LVM_SETIMAGELIST	ListView_SetImageList	Sets one of three image lists of a ListView.
LVM_SETITEM	ListView_SetItem	Sets various attributes of an item.
LVM_SETITEMCOUNT	ListView_SetItemCount	Prepares the control to receive a relatively large quantity of new items.
LVM_SETITEMPOSITION	ListView_SetItemPosition	A given item is moved to a predetermined position in the small and large icon view.
LVM_SETITEMPOSITION32	ListView_SetItemPosition32	Enhanced version of LVM_SETITEMPOSITION with 32-bit coordinates.
LVM_SETITEMSTATE	ListView_SetItemState	Sets the current status of an item.
LVM_SETITEMTEXT	ListView_SetItemText	Sets the text of an item.
LVM_SETTEXTBKCOLOR	ListView_SetTextBkColor	Sets the background color for text in the ListView.
LVM_SETTEXTCOLOR	ListView_SetTextColor	Sets the foreground color for the text in the ListView.
TVM_SORTITEMS	ListView_SortItems	Sorts the items on the basis of sorting function specified by the caller.
LVM_UPDATE	ListView_Update	Updates the view of the control.

ListView notifications

The notifications of a ListView control are always sent to the parent window using a WM_NOTIFY message. You get the control ID of the ListView in the wParam, so you can assign the message. You can only identify the individual notifications with the structure of the NMHDR type, to which the lParam points. Some ListView notifications define their own structures. lParam points to these structures on receiving the WM_NOTIFY message. This occurs although the type NMHDR always appears as the first field of the structure, identifying the sender and the destination of the message.

```
typedef struct {
    HWND    hwndFrom;           // Window handle of the sender
    UINT    idFrom;             // Control ID of the sender
    UINT    code;               // the message code
} NMHDR;
```

The window handle of the sending control is named in the first field, while its control ID is named in the second field. You will also find the control ID in the wParam when you receive the WM_NOTIFY message. Finally, you will find the message code of the notification in the code field. This message code identifies the meaning of the message and determines the parent's response to the message. The following table shows the notifications sent by the ListView control.

A ListView sends LVN messages (List View Notification) to its parent window so the parent is informed about events	
LVN message	Meaning
LVN_BEGINDRAG	Indicates the start of a drag operation with the primary (left) mouse button.
LVM_BEGINRDRAG	Indicates the start of a drag operation with the secondary (right) mouse button.
LVN_BEGINLABELEDIT	Indicates the editing of an item text, parent can refuse.
LVN_COLUMNCLICK	Signals that the passed column was clicked.
LVN_DELETEALLITEMS	Indicates to the parent window that all items in the ListView were deleted.
LVN_DELETEITEM	Indicates to the parent window that a given item was deleted.
LVN_ENDLABELEDIT	Signals the end of the editing of a text item.
LVN_GETDISPINFO	Causes the parent window to return the attributes of an item, so that it can be displayed within the ListView.
LVM_INSERTITEM	Informs the parent that a new item was inserted.
LVN_ITEMCHANGED	Indicates to the parent window that a given item was changed.
LVN_ITEMCHANGING	Indicates to the parent window that an item is to be changed. The parent can refuse this.
LVN_KEYDOWN	Signals the pressing of a key.
LVN_SETDISPINFO	Signals the parent that an item was edited and that it must back up the new text.

Generating a ListView during runtime

Like most other controls, ListViews are generated during runtime by calling `CreateWindowEx()`. This is how you obtain the window handle that you'll use from that point to access the control. Most important, however, you use the call to define the window style of the control by combining the various LVS constants in the `dwStyle` parameter, which designate the appearance and the operation of the control generated. The following partial program code generates, for example, a new ListView control that first appears in the icon view, the icons being automatically aligned along the upper edge of the control. That is taken care of by the specified LVS constants.

```

HWND hListView;

InitCommonControls();
hListView = CreateWindowEx(0,                                // Initialize common controls
                          WC_LISTVIEW,                       // Generate ListView window
                          "",                                  // Window class
                          WS_CHILD |                          // Name
                          WS_VISIBLE |                        // Child window
                          LVS_ICON |                          // Immediately visible
                          LVS_ALIGNLEFT |                     // Icon view
                          LVS_ALIGNTOP,                       // Set alignment
                          x,                                  // Position
                          y,                                  // Dimension
                          cx,
                          cy,
                          hParent,                            // Parent window
                          (HMENU)IDC_LISTVIEW,               // ListView Command ID
                          g_hInstance,                        // Instance
                          NULL);                              // User data

if (!hListView)                                             // Could window be generated?
{
    // Error
}

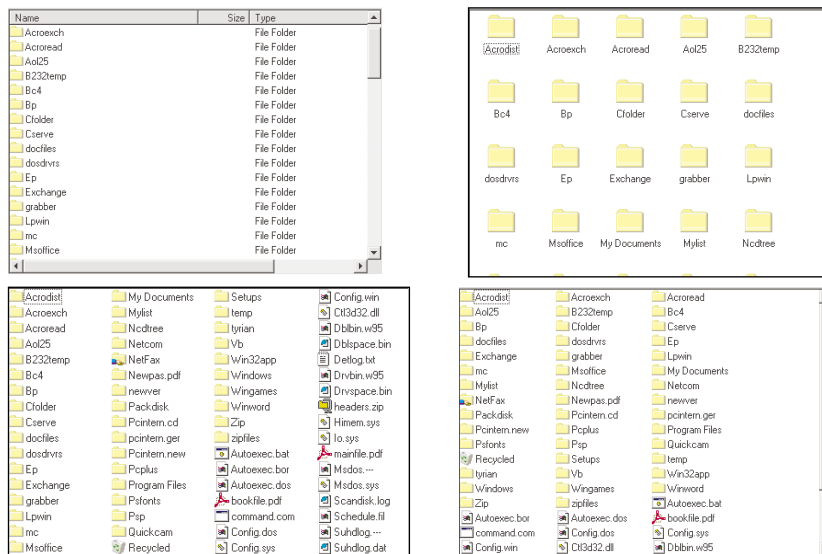
```

The view modes

A new buzz word here is *icon view*. The view mode of the control is specified initially when it is generated but can be changed at any time. Four view modes are available: large icon view with large pictures, “small icon” view with icons of reduced size as well as list view and report view.

As the following illustration shows, large icon view, small icon view and list view are similar. Report view has a much different appearance. The four views are also predestined for different fields of application. The first three views we mentioned help with the graphic representation of given container, the individual elements consisting of an image and a brief text. Online services, like MSN and Compuserve, use this control in their browsers to display the contents of a forum. Report view, on the other hand, looks like a multicolumn list box where each item occupies a line with several columns. This presentation is therefore used when you want to display tables with items containing more information than a simple text.

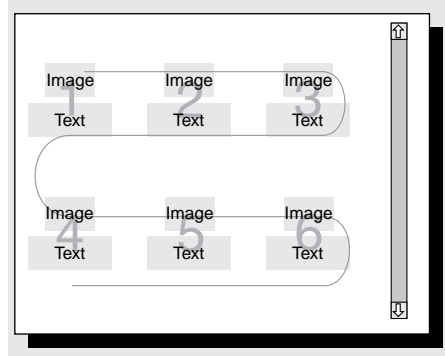
The four view modes of a ListView control using the Explorer as an example



LVS_SMALLICON

The (default) icon view

Display sequence in default icon view



In large icon view, you're not dealing with a list in the proper sense. What is most apparent to the user are above all the images (icons), with a brief text appearing under each one. The text can have one or more lines according to the length of the text. The items can be arbitrarily arranged by the program, and a logic can be implemented with the support of the control, permitting the user to move the items using Drag and Drop. By specifying appropriate LVS constants, it's also possible to have the control arrange the symbols automatically.

Initially, the arrangement of the individual images results from the sequence in which they were inserted

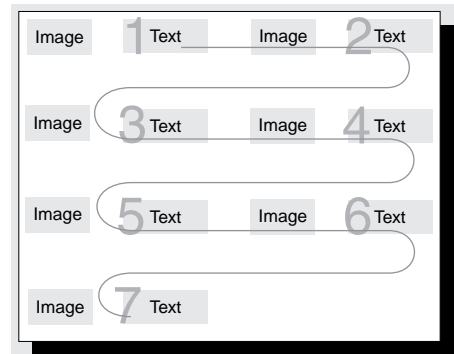
in the control during runtime (from left to right and from top to bottom). The width of the items and the width of the control determine how many items are arranged in a line. Whenever possible, only just so many entries are inserted in a line that they do not extend beyond the edge. Therefore, a horizontal scroll bar is not required. If the space in the client area of the control is not adequate for the display, the control gets a vertical scroll bar.

LVS_SMALLICON

The icon view with small icons

This display resembles the default icon view in many respects. The most noticeable difference is that the icons here are smaller and the text appears to the right of the icon. Icons here can also be arranged as desired. As with the default icon view, the items are arranged initially from left to right and from top to bottom. The main advantage of this mode over default icon view is that more items fit into the client area of the control. This is because the smaller icons require less room.

Display sequence in small icon view

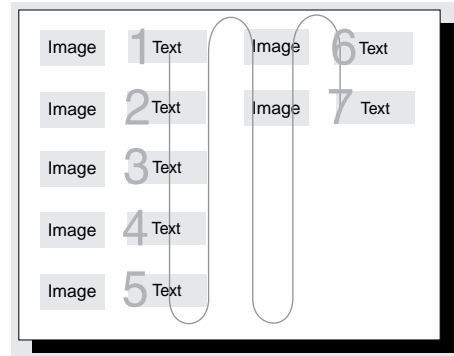


LVS_LIST

List view

List view resembles small icon view because the text is also displayed on the right, next to the small icon. Nevertheless, there are two important differences:

Display sequence in the list view



1. The control determines the arrangement of the items so each item follows the next with at the preset distance. Therefore, arbitrary positioning of the items, as in the case of the icon views, is not possible.
2. The display sequence is different. The views are first arranged from top to bottom and then from left to right. Only so many items are positioned above one another, so that no vertical scroll bar is required. If the control contains more items than can be displayed in the client area, they will extend beyond the righthand edge of the control, so that a horizontal scroll bar appears.

LVS_REPORT

Report view

Everything is different in report view. In this view, the image of an item is not in the foreground. Instead, the individual columns containing various data on the item appear in the foreground. If desired, you can have a column heading appear above the columns. Users can change the column width manually with this heading. Moreover, users can also sort the items according to the respective column criterion by clicking a column. However, this requires the support of the program.

Display sequence in the list view

	Main heading	Set Item	Set Item	Set Item
	Column heading	Column heading	Column heading	Column heading
1	Image Text	Text	Text	Text
2	Image Text	Text	Text	Text
3	Image Text	Text	Text	Text
4	Image Text	Text	Text	Text
5	Image Text	Text	Text	Text
6	Image Text	Text	Text	Text

The (small) icon of the item appears in the first column. Its text appears to its right. The texts in the additional

columns are designated as “subitems” and have to be loaded by the program separately. Each item occupies one line with the individual items arranged from top to bottom, according to their order of insertion.

Although the control of this view mode is based on the same fundamentals as the other three modes, several LVM messages act specifically on this mode. For example, there is a message for the setting of the columns or loading the subitems. For more information on this view mode, see the section about “Working in Report View” later in this chapter.

The additional window styles

Besides the various constants for setting the view mode, additional window styles are available to influence the appearance and the operation of the ListView control. These are summarized in the following table (and explained later in more detail):

The LVS constants (ListView styles) determine the appearance and the operation of a control.	
Constant	Meaning
LVS_ICON	The control appears in icon view with large icons.
LVS_SMALLICON	Icon view with small icons.
LVS_REPORT	Report view.
LVS_LIST	List view (possibly with subitems).
LVS_SINGLESEL	Only one item at a time can be selected.
LVS_SHOWSELALWAYS	Always display the selection, even if the control does not have the focus.
LVS_SORTASCENDING	On insertion sort in ascending order .
LVS_SORTDESCENDING	On insertion sort in descending order.
LVS_SHAREIMAGELISTS	Control shares the image list with other controls.
LVS_NOLABELWRAP	Displays the label in icon view in only one line.
LVS_AUTOARRANGE	Arranges the icons automatically in both icon views.
LVS_EDITLABELS	The user can edit the label of an item.
LVS_NOSCROLL	No scrolling possible.
LVS_ALIGNTOP	The icons are automatically positioned at the top of the client area in both icon views.
LVS_ALIGNLEFT	The icons are automatically arranged at the left border.
LVS_NOCOLUMNHEADER	Don't display column headings.
LVS_NOSORTHEADER	Column headings cannot be clicked for sorting.

Switching between the various styles

To change the display mode or that of one of the other LVS flags later, use the SetWindowLong() API function from the USER32.DLL. It's possible by using USER32.DLL to change a series of settings which Windows retains for each window. For example, you can change the address of the message procedure and the current window style. The window style is represented by the GWL_STYLE constant. Specify this constant as the second parameter when calling SetWindowLong().

```

LONG SetWindowLong(
    HWND hWnd,                // Handle of the window addressed
    int nIndex,                // Offset of LONG, one of the GWL constants
    LONG dwNewLong             // New value
);

```

In addition, specify the window handle of the window addressed in the first parameter and, in the last parameter, the new value for the window style. Because you want to change only one specific flag in the window style and keep all the others, we recommend that you first determine the current window style, insert or remove the desired flags and then establish the new style in the window by calling SetWindowLong().

Determine the window style with the GetWindowLong() API function. This function likewise expects the window handle and the offset of the desired LONG in the form of the GWL constant as arguments. In our case, once again, this is GWL_STYLE. The function then returns the current window style as the result.

```

LONG GetWindowLong(
    HWND  hWnd,                                // Window handle
    int    nIndex                               // One of the GWL constants
);

```

The following partial program code shows how a window style is changed in practice. In this case, the current window style is first read from the ListView control with the hListView handle and then the LVS_EDITLABELS flag switched over for the editing of the item labels. Then, the new style is then permanently written in the window structure using SetWindowLong().

```

// Determine old style and switch -----
st.styleOld = GetWindowLong( hListView, GWL_STYLE );
st.styleNew = st.styleOld;

if( st.styleNew & LVS_EDITLABELS )
    st.styleNew &= ~LVS_EDITLABELS;
else
    st.styleNew |= LVS_EDITLABELS;

// Set new style -----
SetWindowLong() hListView, GWL_STYLE, st.styleNew );

// and notify control of change -----
SendMessage( hListView,
              WM_STYLECHANGED,
              ( WPARAM ) GWL_STYLE,
              ( LPARAM ) &st );

```

To be on the safe side, a WM_STYLECHANGED message is also sent to the control. The control is notified that the window style has changed and that it must respond accordingly. The Microsoft documentation claims that SetWindowLong() automatically sends a corresponding report to the control. However, it doesn't always seem to work satisfactorily in practice. That's why we play it safe and send this report explicitly.

Adding and deleting items

The items in a ListView must be generated before they can be displayed. Use the LVM_INSERTITEM message for this purpose. Specify 0 in the wParam, specify the address of an LV_ITEM structure in the lParam. Like a TV_ITEM in a TreeView, the LV_ITEM structure represents an item in a ListView and is inserted within the frame of a multitude of messages which act on the ListView items.

```

#define ListView_InsertItem(hwnd, pItem) \
    (int)SendMessage( (hwnd), LVM_INSERTITEM, \
                      0, (LPARAM)(const LV_ITEM FAR*)(pItem) )

```

After sending the message, the index of the new item is returned or -1 if the item could not be inserted. Handle the index with caution if you want to store it for accessing the item later. The index reflects the current position of the item, which can change during execution. Examples of this are inserting the items in the two icon views or performing a sorting operation. Unlike a handle, which accompanies an object during its entire life, the index is therefore volatile. Don't rely on it to remain the same. The LV_ITEM structure makes certain the link with the item is maintained without having to access the original index.

The index of an item can change when the next additional item is inserted if you specified either the LVS_SORTASCENDING or LVS_SORTDESCENDING flag in the window style of the control. If that is the case, a new item will not be simply added to the end of the list, but sorted in based on its label. The result is that the items automatically appear after insertion in alphabetically ascending (LVS_SORTASCENDING) or descending (LVS_SORTDESCENDING) order.

The LV_ITEM structure

The LV_ITEM structure determines the appearance and attributes of an item and must be specified in the lParam by the senders of the LVM_INSERTITEM message.

```
typedef struct {
    UINT    mask;                // Combination of the LVIF constants
    int     iItem;               // Number of the item addressed
    int     iSubItem;            // Number of the subitem
    UINT    state;               // Combination of the LVIS flags of the setting
    UINT    stateMask;           // Combination of the LVIS flags for masking
    LPSTR    pszText;            // Pointer to C-string with text
    int     cchTextMax;          // Size of the text buffer
    int     iImage;              // Number of the item
    LPARAM   lParam;             // Arbitrary value of the caller
} LV_ITEM;
```

mask

A concept we're already familiar with from TV_ITEM is also employed with LV_ITEM. We're talking about using constants in the mask field to validate initialized fields of the item.

When creating a new item, you have to combine all four LVIF constants: LVIF_TEXT, LVIF_IMAGE, LVIF_PARAM and LVIF_STATE and specify at least 0 in the associated fields to initialize them.

LVIF flags (List View Item Flags) characterize the fields initialized in a LV_ITEM structure.	
Constant	Meaning
LVIF_TEXT	The LV_ITEM.pszText field is initialized.
LVIF_IMAGE	The LV_ITEM.iImage field is initialized.
LVIF_PARAM	The LV_ITEM.lParam field is initialized.
LVIF_STATE	The LV_ITEM.state field is initialized.

iItem and iSubItem

The iItem and iSubItem fields have to do with an item's number. In this context, you need to know that you don't use handles to address the items of a ListView. Instead, you address the items with indexes, which the items receive during insertion. That's why the two fields are not initialized before sending a LVM_INSERTITEM message. However, you do initialize them later to influence an existing item (for example, using the LVM_SETITEM message). The iItem field designates the index of the item, while iSubItem designates the index of the associated subitem. You will find subitems only in report view. We'll cover this topic in greater detail later in this chapter. Until then, we'll concentrate on the iItem field and ignore the iSubItem field. Simply specify a 0 for iSubItem.

state and stateMask

These fields also follow the model in TV_ITEM. The individual bits in state represent the various states of an item as one of the LVIS constants. Use stateMask to name the flags you want to act on if a flag is to be set or read out in the context of an LVM message. When setting the attributes, insert the corresponding flags in the stateMask field to enable them. Specify a flag in stateMask that does not appear in state to disable the corresponding attribute.

LVIS constants (List View Item State) represent an item's attributes.	
Constant	Meaning
LVIS_FOCUSED	The item has the focus.
LVIS_SELECTED	The item is selected.
LVIS_CUT	The item is highlighted for removal.
LVIS_DROPHILITED	The item is designated as the possible target of a drag and drop operation.
LVIS_OVERLAYMASK	For masking by an overlay image.
LVIS_STATEIMAGEMASK	For displaying a state image next to the image.

First, the various flags show the current status of an item, indicating whether it has the focus or is selected. They also determine whether the item is highlighted as the target of a potential Drag and Drop operation and whether the image of the item will receive an overlay or a state image (or both). We'll examine both operations more closely in the following section.

pszText and cchTextMax

These fields are used to manage an item's text. When an item is created or modified, pszText is initialized with the address of the C-string containing the desired text. cchTextMax has nothing to do with this and is required only when the text is read out using the LVM_GETITEM message. Before being called, pszText must be placed in one of the buffers prepared by the caller and the length of this buffer in characters must be specified in cchTextMax. In this way, the control ensures that the buffer will not overflow because it is too short for the text.

ilImage

The image of the item is stored here as an index in an image list assigned to the control. Ignore this field as long as an image list isn't set because only text is to appear, with no images. We'll talk more about the images of a ListView in the following section.

IParam

This field gives the owner of a ListView the option of storing its information with an item. This makes it possible to maintain the link between the program code and the item. First, the value entered here can be used to search for items using the LVM_FINDITEM message. Besides that, the value is included in the notifications of the control (LVN messages). A program can use this field, for example, to place a pointer to a program-internal data structure in it or use it as its own index which, unlike the index returned by the control, remains constant.

Insertion in practice

Thus, an entire series of constants and fields is also involved when you want to create new items in a ListView. The following partial code illustrates once more the events during the creation of a ListView and the insertion of items. In concrete terms, a ListView is generated in the list view and twenty six items are created with the letters A through Z. It is apparent from the result returned by the SendMessage() function whether the desired item can actually be inserted. So, the return value of -1 stands for an error. All other values represent the index of the new item in the ListView.

```

HWND      hListView;                                // Handle of the ListView
LV_ITEM   lvi;                                     // Structure for the insertion of items
char      szAlphabet[ 2 ];                          // Text buffer
int        iItem;                                  // Index of the added item
int        i;                                       // Counter

InitCommonControls();                               // Initialize CommonControls

hListView = CreateWindowEx(0,                        // Generate window
                          WC_LISTVIEW,              // Window class
                          "",                        // Name
                          WS_CHILD |                // Child window
                          WS_VISIBLE |              //
Immediately visible
                          LVS_LIST,                  // Icon representation
                          x,                          // Position
                          y,                          //
                          cx,                        // Dimension
                          cy,                        //
                          hParent,                    // Parent window

```

```

                                (HMENU)IDC_LISTVIEW,      //ListView Command ID
                                g_hInstance,              // Instance
                                NULL);                  // User data
if (hListView)                                // Could window be generated?
{
    ListView_SetItemCount( hListView, 26 );           // Make room for 26 entries

    szAlphabet[ 1 ] = '\0';

    // Prepare LV_ITEM structure. This needs to be done only once, because
    // items are generated which differ from each other only in the text.
    lvi.mask          = LVIF_TEXT;                  // pszText and cchTextMax are valid
    lvi.iSubItem       = 0;                        // Generate new main item(SubItem = 0)
    lvi.pszText        = szAlphabet;                // Set address of the text buffer
    lvi.cchTextMax     = sizeof( szAlphabet );       // Size of the text buffer

    for( i = 0; i < 26; i++ )                      // Run through each letter
    {
        szAlphabet[ 0 ] = 'A' + i;                 // Change only text buffer
        iItem = ListView_InsertItem(hListView, &lvi); // Generate item

        if( iItem == -1 )                          // Item generated??
        {
            // Error
        }
    }
}

```

You probably noticed the `ListView_SetItemCount()` function in the above partial code, which is called before the first insertion of an item. This macro uses `SendMessage()` to send the `LVM_SETITEMCOUNT` message to the `ListView` control.

```

#define ListView_SetItemCount(hwndLV, cItems) \
    SendMessage( (hwndLV), LVM_SETITEMCOUNT, (WPARAM)cItems, 0)

```

Using this message is optional and is recommended when several items are to be inserted in rapid sequence. Rather than being forced to enlarge the internal memory area again with each new insertion, the control automatically sets the specified size when the message is received. This helps to speed up the later insertion operations (`LVM_INSERTITEM`). Specify the total number of entries in the `wParam` as the argument of the message. The control is not so selective on that point, allowing the actual number to be smaller or larger than that indicated. Thus the information serves merely as a guideline. If the memory allocated on the basis of this information is used up by the following `LVM_INSERTITEM` messages, new memory will simply be added to the allocation with the next `LVM_INSERTITEM`.

Notifying the parent window

With the insertion of an item, the parent window of the control receives an `LVN_INSERTITEM` message using a `WM_NOTIFY` message to inform it that a new item was inserted. This message has no access possibilities and simply gives the parent the opportunity to take note of the action.

```

typedef struct {
    NMHDR    hdr;                // The usual header of a WM_NOTIFY message
    int      iItem;              // Number of the new item inserted
    int      iSubItem;           // 0
    UINT     uNewState;          // 0
    UINT     uOldState;          // 0
}

```

```

        UINT    uChanged;                                // 0
        POINT   ptAction;                                // 0
        LPARAM  lParam;                                  // 0
    } NM_LISTVIEW, FAR *LPNM_LISTVIEW;

```

The only two fields initialized in this structure with significant values are `hdr` with the usual notification header and `iltem` with the number of the new item inserted. If the parent wants information on this item, it can use the `LVM_GETITEM` message.

Deleting items

Use the `LVM_DELETEITEM` message to delete an item. This message is copied by the macro `ListView_DeleteItem`.

```

#define ListView_DeleteItem(hwnd, i) \
    (BOOL)SendMessage((hwnd), LVM_DELETEITEM, (WPARAM)(int)(i), 0L)

```

The index of the item to be deleted is specified in the `wParam`. Additional information with this message is not required. The return value is `TRUE` if this item is removed as desired. If the deleted item was visible in the client area of the control, it will be redrawn automatically.

The parent window detects the deletion of an item by responding to the `LVN_DELETEITEM` message in its message procedure. It receives the message from the control as a part of a `WM_NOTIFY` message when an item is deleted. The entry to be deleted is therefore represented by a structure of the `NM_LISTVIEW` type, whose address is delivered to the message by the control in `lParam`. The usual header with a structure of the `NMHDR` type appears at the beginning. It contains the sender and the actual notification code (`LVN_DELETEITEM` in this example).

```

typedef struct {
    NMHDR    hdr;                                // The usual header of a WM_NOTIFY message
    int      iItem;                               // Number of the item in question, -1 is unspecified
    int      iSubItem;                            // Number of the subitem (report display only),
                                                // otherwise 0
    UINT     uNewState;                           // Old status, LVIS constant combination
    UINT     uOldState;                           // New status from LVIS constants
    UINT     uChanged;                            // LVIS constants which have changed
    POINT    ptAction;                            // Coordinates, for drag notifications only
    LPARAM   lParam;                              // lParam from LV_ITEM structure
} NM_LISTVIEW, FAR *LPNM_LISTVIEW;

```

The `iltem` field returns the number of the item concerned, in report view you also get the associated number of the subitem in `iSubItem`. The notification determines if the remaining fields are filled. After all, `LVN_DELETEITEM` doesn't use this structure exclusively. In each case, however, `lParam` field is still passed as the owner loaded it during generation of the item. If you're not satisfied with deleting individual items and want to delete everything, use the `LVM_DELETEALLITEMS` message or the associated macro `ListView_DeleteAllItems()`.

```

#define ListView_DeleteAllItems(hwnd) \
    (BOOL)SendMessage((hwnd), LVM_DELETEALLITEMS, 0, 0L)

```

Aside from the message, no information is required in the `wParam` and `lParam`. The parent quickly receives an `LVN_DELETEALLITEMS` message, but then the items are all gone. Along with the notification, you receive a pointer to an `NM_LISTVIEW` structure again in the `lParam`. The `iltem` field is initialized with `-1` to indicate all items were deleted.

The images of a ListView

The images of the individual items in a `ListView` stand out even more than the texts. As in the case of `TreeView`, the individual items don't store the image itself, but use an `Image List` for this purpose, which is coupled to the `ListView`. So, the images appear as icons and are drawn transparently in the foreground. When inserting an item, you don't have to specify the image in the `LV_ITEM` structure, but only the index of the image in the `Image List`. Therefore, several items can use the same image without having to store it several times in memory.

The Image List also determines the size of the images. The ListView control does not limit the size of the images to that of Explorer icons. The images can be significantly larger. Therefore, displaying images requires the addition of an Image List using the LVM_SETIMAGELIST message or the associated ListView_SetImageList() macro.

```
#define ListView_SetImageList(hwnd, himl, iImageList) \
    (HIMAGELIST)(UINT)SendMessage((hwnd), LVM_SETIMAGELIST, \
    (WPARAM)(iImageList), \
    (LPARAM)(UINT)(HIMAGELIST)(himl))
```

The message expects the handle of the Image List in the lParam, while it expects one of the three LVSIL constants in the wParam. They represent the three image lists which can be linked to a ListView control. Each of them has a very special job. The Image List represented by LVSIL_NORMAL operates in large icon view, while the one represented by LVSIL_SMALL is used in small icon view. Depending on the display mode, the ListView control switches between these two Image Lists automatically. No one in that case specifies the size of the images. If you wanted, you could even make the images in the Image List of LVSIL_SMALL larger than those under LVSIL_NORMAL. Keep in mind that the sequence of the images and their motifs must be identical, for the index of the image in LV_ITEM.iImage does not get switched.

The issue of whether you really need to set two different Image Lists depends on the view in which you want to present the ListView to the user and whether he is to have the option of switching independently between the different views. If the display remains limited to one specific mode, you need to initialize only that Image List associated with it.

You'll need the third Image List with state images only if you want to display state images along with the individual images. We demonstrated this for you in earlier in the chapter with TreeView. You can use state images to display a graphic status next to an item's image. For example, you could use a state image to indicate an item is active or is included among a very special type of items. The size of the images in the Image List specified here also determines, in turn, the size of the state images. We will now look at how you set the state image of a given item and the index in the Image List of the state images.

LVSIL constants (List View Set Image List) for setting an Image List	
Constant	Meaning
LVSIL_NORMAL	Image List for default (large) icon view.
LVSIL_SMALL	Image List for small icon view, list view and report view.
LVSIL_STATE	Image List for state images.

After sending the LVM_SETITEMLIST message, you'll also receive the handle of the Image List set up to that point (providing one was even set). NULL is returned if this is not the case. To display the images in the same size as the icons in the Explorer, use the API function GetSystemMetrics() to determine the size of the icons in the default and small versions. Then scale your images accordingly before forwarding them to the ListView control.

State images and overlays

Along with state images, it's also possible to mask images with overlays, as they are defined using the ImageList_SetOverlayImage() function. The overlays must originate from the same Image List as the image itself. A maximum of one overlay and one state image can be added to each image (and therefore to each item).

However, our starting point is not the LV_ITEM.iImage field with the number of the (regular) image. Instead, we start with the state field. You have to add the result of the INDEXTOOVERLAY() macro to the contents of this field. We have to pass the macro a number from 1 to 4, for one of the four possible overlays. If you assign the value 0 to the macro, the current overlay image will again be hidden.

Continue similarly with INDEXTOOVERLAYMASK(). This is a macro whose result must likewise be added to the value of state field. It receives the value of the state image as a number between 1 and 15. Once again, specifying 0 hides the current state image. By looking closely at the two macros, it becomes apparent they merely shift the value passed to them a few bit-positions to the left. The macros do this to insert the value in places in the state field that aren't being used by the actual state flags (LVIS_SELECTED, etc.). The overlay number migrates into bits eight through ten, the state image into bits twelve through fifteen.

The following partial program code shows how a new item is inserted with the second overlay added and the item linked to state image number 5.

```

LV_ITEM lvi;
int      iItem;
lvi.mask    = LVIF_TEXT | LVIF_STATE;           // Text and state valid
lvi.iSubItem = 0;                               // Generate new main item

lvi.state    = INDEXTOSTATEIMAGEMASK( 5 ) |      // StateImage 5
               INDEXTOOVERLAYMASK( 2 );          // OverlayImage 2
lvi.stateMask = LVIS_STATEIMAGEMASK |            // StateImage was declared
               LVIS_OVERLAYMASK;                 // OverlayImage was declared

lvi.pszText   = lpText;                          // Set text
lvi.cchTextMax = strlen( lpText );

iItem = ListView_InsertItem(hListView, &lvi);    // Insert item

```

You can declare overlays and state images immediately when an item is generated or use the LVM_SETITEM method to add them later. The LVM_SETITEM method lets you change all of an item's attributes, including text, image and status. We'll tell you more about this in the next section.

Querying the Image List set

To determine which Image Lists are set in a control, use the LVM_GETIMAGELIST message or the associated macro ListView_GetImageList(). Specify one of the LVSIL constants in the wParam to designate the Image List whose handle you want to determine.

```

#define ListView_GetImageList(hwnd, iImageList) \
    (HIMAGELIST)SendMessage((hwnd), LVM_getimagelist, \
        (WPARAM)(INT)(iImageList), 0L)

```

The handle itself is returned as the function result of SendMessage().

During “destruction” of a ListView control as a response to a WM_DESTROY message, the linked Image Lists are normally also deleted by calling the Image_ListDestroy() function. Under certain circumstances you may want to prevent that, because the Image List is being used at the same time in other controls, or you want to use them later on in the program. So, you can suppress the automatic deletion of the Image Lists. This is done in an unexpected place: You need to specify a corresponding flag named LVS_SHAREIMAGELISTS in the window attribute of the control to prevent it from destroying more than absolutely necessary.

Querying and setting items

To change an item's attributes later, use the LVM_SETITEM message or the associated ListView_SetItem() function macro. The message must be accompanied in the lParam by the address of an LV_ITEM structure in which the caller has deposited the changed attributes. wParam is not important.

```

#define ListView_SetItem(hwnd, pitem) \
    (BOOL)SendMessage((hwnd), LVM_SETITEM, \
        0, (LPARAM)(const LV_ITEM FAR*)(pitem))

```

Specify the attributes to be changed in the mask field of the LV_ITEM structure using a combination of the LVIF constants provided for it (see the following table).

LVIF flags (List View Item Flags) during the change of an item with LVM_SETITEM.	
Constant	Meaning
LVIF_TEXT	The pszText field contains a pointer to the string with the new text of the item.
LVIF_IMAGE	The ilmage field contains the new value for the index in the Image List.
LVIF_PARAM	The lParam field contains a new value to be stored.
LVIF_STATE	A new status is to be set using the state and stateMask fields.

To reset one of the attributes in state, you must consider the interplay described earlier between state and stateMask. Set the flags of the attributes you want to change in stateMask. Then set the respective flags in state only if the associated setting is to be enabled. To change the setting for the overlay image or the state image, specify the associated macros with the values for the two controls in both stateMask and state.

The following partial program code shows how all items of a ListView are run through and marked as selected by switching on the corresponding flag in the state field of the LV_ITEM structure.

```
int iCnt;
int i;

iCnt = ListView_GetItemCount( hListView );           // Determine number of items

for( i = 0; i < iCnt; i++ )                          // Run through all items
{
    LV_ITEM lvi;
    lvi.mask = LVIF_STATE;                            // We are interested only in the status
    lvi.iItem = i;                                    // Index of the item in question
    lvi.iSubItem = 0;                                 // Must be 0 to determine the main item
    if( ListView_GetItem( hListView, &lvi ) )         // Fetch the data of the item
    {
        lvi.state |= LVIS_SELECTED;                  // Selected
        lvi.stateMask |= LVIS_SELECTED;              // Consider status selected in state
        ListView_SetItem( hListView, &lvi );         // Change item
    }
}
```

Notifications when making changes

When an item is being changed, the parent of the ListView control receives two LVN messages. These messages notify the parent of the change and even give the parent the opportunity to prevent the change. In both cases, the WM_NOTIFY structure in lParam is accompanied by the address of an NM_LISTVIEW structure. The NM_LISTVIEW structure specifies the origin of the message and the message itself, but it also describes the item with its new attributes.

```
typedef struct {
    NMHDR    hdr;                // The conventional header of a WM_NOTIFY message
    int      iItem;              // Number of the item in question
    int      iSubItem;           // Number of the subitem (report display only),
                                // otherwise 0
    UINT     uNewState;          // Old status, LVIS constant combination
    UINT     uOldState;          // New status from LVIS constants
    UINT     uChanged;           // LVIS constants which have changed
    POINT     ptAction;           // In this case 0
    LPARAM    lParam;            // lParam from LV_ITEM structure
} NM_LISTVIEW, FAR *LPNM_LISTVIEW;
```

Before the change is completed, the parent receives an LVN_ITEMCHANGING message. This message gives the parent the option of accepting the change or rejecting it. For example, in this way you can prevent the selection of specific items. To suppress the change, all the message procedure of the parent needs to do is return the value TRUE. If the message from the

parent is not evaluated, or explicitly returns FALSE, the change will occur as desired. Soon thereafter the parent receives an LVN_ITEMCHANGED message with the same NM_LISTVIEW structure in the lParam indicating the change.

Querying with LVM_GETITEM

The counterpart to LVM_SETITEM is the LVM_GETITEM message. Again, the address of an LV_ITEM structure must be passed in the lParam. This time, the caller only needs to initialize the mask field of the structure along with the iItem and iSubItem fields for designation of the item, using the LVIF constants which point to the attributes to be determined. For example, use LVIF_IMAGE, if the number of the image representing the item to the user is to appear in the LV_ITEM.iImage field after the return.

LVIF flags (List View Item Flags) for querying an item using LVM_GETITEM.	
Constant	Meaning
LVIF_TEXT	Return the text of the item in the buffer to which pszText points.
LVIF_IMAGE	Return image of the item in iImage.
LVIF_PARAM	Return lParam of the item.
LVIF_STATE	Return current status of the state field.

To determine the text of an item, it's first necessary to prepare a corresponding buffer before the call. The control will copy the name into this buffer. The TV_ITEM.pszText field must be loaded with the address of this buffer before the message is sent, cchTextMax with its size in characters. The associated macro for this message is ListView_GetItem():

```
#define ListView_GetItem(hwnd, pItem) \
    (BOOL)SendMessage((hwnd), LVM_GETITEM, \
        0, (LPARAM)(LV_ITEM FAR*)(pItem))
```

Exception, text

Use two special messages, LVM_SETITEMTEXT or LVM_GETITEMTEXT, to save some work if you only want to read or determine the text of an item. In both cases, pass the number of the addressed item in wParam, but do not declare it in the iItem field of the LV_ITEM whose address is also expected in lParam with these two messages. The iSubItem field is only used when querying a subitem in report view to determine the number of the subitem to be queried.

The message takes care of setting the mask field for you. However, the pszText field must be initialized before the call. Use the address of the C-string containing the new text to initialize LVM_SETITEMTEXT and use the address of the buffer into which the text is to be copied for LVM_GETITEMTEXT. In keeping with that, LV_ITEM.cchTextMax must also be initialized. So you won't really save much work compared with the standard LVM_GETITEM message.

The two macros also turn out to be complicated, because you are trying to get by without an LV_ITEM structure. However, because this structure is required for the call, the macro creates a new block of code, in which the required LV_ITEM structure is included as a local variable and initialized with the transmitted parameters and the address of the C-string.

```
#define ListView_SetItemText(hwndLV, i, iSubItem_, pszText_) \
{ LV_ITEM _ms_lvi;\
  _ms_lvi.iSubItem = iSubItem_;\
  _ms_lvi.pszText = pszText_;\
  SendMessage((hwndLV), LVM_SETITEMTEXT, \
      (WPARAM)i, (LPARAM)(LV_ITEM FAR *)&_ms_lvi);\
}
```

When we are confronted with such macros, two questions arise: Does it really makes sense to use them? Aren't we generating too much hidden code? ListView_GetItem() doesn't look any better especially since cchTextMax must also be specified here as an additional argument. At least it's a bit more transparent because its LV_ITEM structure can also be readily built by hand.

```
#define ListView_GetItemText(hwndLV, i, iSubItem_, pszText_, cchTextMax_) \
{ LV_ITEM _ms_lvi;\
  _ms_lvi.iSubItem = iSubItem_;\
```

```

    _ms_lvi.cchTextMax = cchTextMax;\
    _ms_lvi.pszText = pszText;\
    SendMessage((hwndLV), LVM_GETITEMTEXT, \
                (WPARAM)i, (LPARAM)(LV_ITEM FAR *)&_ms_lvi);\
}

```

It's not much better with the LVM_SETITEMSTATE and LVM_GETITEMSTATE messages, which you can use in the same way to get direct access to the status of an item. LVM_GETITEMSTATE and the associated macro are really practical, because you only need to specify the number of the item in wParam and, in lParam, the mask with the desired flag as a combination from the LVIS constant. The current status is returned immediately as the function result of SendMessage(). Only the flags specified in lParam for the call were examined, so evaluation should remain limited to these flags.

```

#define ListView_GetItemState(hwndLV, i, mask) \
    (UINT)SendMessage((hwndLV), LVM_GETITEMSTATE, \
        (WPARAM)i, (LPARAM)mask)

```

It gets crazy again in the case of LVM_SETITEM. Besides the number of the item, two parameters for state and stateMask are required. However, they cannot be accommodated together in lParam. Once again, you need an LV_ITEM structure in which the two fields are correspondingly initialized. So the only work the macro saves the caller from performing is the setting of the mask field. The macro attempts to take over the declaration and initialization of the LV_ITEM structure for the caller, but that does not necessarily make the whole thing more transparent.

```

#define ListView_SetItemState(hwndLV, i, data, mask) \
{ LV_ITEM _ms_lvi;\
  _ms_lvi.stateMask = mask;\
  _ms_lvi.state = data;\
  SendMessage((hwndLV), LVM_SETITEMSTATE, \
              (WPARAM)i, (LPARAM)(LV_ITEM FAR *)&_ms_lvi);\
}

```

Setting colors and fonts

The ListView offers its owner the option of setting the background color of the client area as well as the foreground and background color of the text. Fortunately, the traditional WM messages are not needed here because six messages are available for that purpose. LVM_SETBKCOLOR and LVM_GETBKCOLOR act on the background color of the client area. LVM_SETTEXTCOLOR, LVM_GETTEXTCOLOR, LVM_SETTEXTBKCOLOR and LVM_GETTEXTBKCOLOR have to do with the font color of the text and its background color. The colors are exchanged as a COLORREF (RGB value in long) using lParam or the function result of SendMessage(). The wParam has no role in these messages.

```

#define ListView_GetBkColor(hwnd) \
    (COLORREF)SendMessage((hwnd), LVM_GETBKCOLOR, 0, 0L)
#define ListView_SetBkColor(hwnd, clrBk) \
    (BOOL)SendMessage((hwnd), LVM_SETBKCOLOR, \
        0, (LPARAM)(COLORREF)(clrBk))
#define ListView_SetTextColor(hwnd, clrText) \
    (BOOL)SendMessage((hwnd), LVM_SETTEXTCOLOR, \
        0, (LPARAM)(COLORREF)(clrText))
#define ListView_GetTextColor(hwnd) \
    (COLORREF)SendMessage((hwnd), LVM_GETTEXTCOLOR, 0, 0L)
#define ListView_GetTextBkColor(hwnd) \
    (COLORREF)SendMessage((hwnd), LVM_GETTEXTBKCOLOR, 0, 0L)
#define ListView_SetTextBkColor(hwnd, clrTextBk) \
    (BOOL)SendMessage((hwnd), LVM_SETTEXTBKCOLOR, \
        0, (LPARAM)(COLORREF)(clrTextBk))

```

Setting the font

Like the other common controls, ListView has no special message available for setting the font. Instead, use the default message to set a new font, WM_SETFONT. When sending the message, specify the handle of the font in wParam. TRUE or FALSE is required in the low-word of lParam. Use TRUE if you want the control to begin redrawing immediately with the new font, or use FALSE to have the new font used only with the next redrawing of an item.

For example, the following partial program code shows how the default dialog font of Windows 95 is set in the ListView with the hListView handle.

```
// Set default Win95 dialog font
SendMessage( hListView,
             WM_SETFONT,
             ( WPARAM )GetStockObject( DEFAULT_GUI_FONT ),
             MAKELONG( TRUE, 0 ) );
```

Browsing the selected items and searching for items

Use the LVM_GETNEXTITEM and LVM_FINDITEM messages to browse through the items in a ListView, for example, to back them up or to determine which items are selected. LVM_GETNEXTITEM is designed more for browsing items (as in a directory), whereas LVM_FINDITEM searches for a specific item validated by its text or its lParam.

```
#define ListView_GetNextItem(hwnd, i, flags) \
    (int)SendMessage( hwnd, LVM_GETNEXTITEM, \
                      (WPARAM)(int)(i), MAKELPARAM((flags), 0) )
```

Use the ListView_GetNextItem() macro to browse through the items. The number of the starting item is passed in wParam. lParam receives a flag that determines the direction of the search and the type of item that the control returns as the next item. One of five LVNI constants specifies the search direction.

The lower four constants are aimed primarily at icon and list views, where the various items are presented in a somewhat ordered matrix and you can search for the next item in all four directions. To browse through the items on the basis of their index, regardless of their arrangement, specify LVNI_ALL. The starting point for the search is always the item whose number is entered in wParam of the message. To begin with the item at the top of the list, enter -1.

Besides the LVNI constants we've mentioned, you can specify other LVNI constants in lParam. These constants limit the entries returned to a specific criterion. For example, specify one of these constants if you only want to read out the items that are selected. If the search is successful, SendMessage() returns the index of the item found as the function result, otherwise it returns -1, if no other item with the desired attributes could be found in the desired direction.

The following partial program code shows how to browse through a given ListView in index sequence to get all the selected items. Specify -1 in wParam to start with the first item in the list. Then continue sending LVM_GETNEXTITEM until you get -1 as the function result. Specify the function result of the preceding call in wParam so you always start with the previously returned item. Then use ListView_GetItemText() to get the text of each found item and use printf() to output the text to the console.

LVNI constants (List View Next Item) for setting the search direction	
Constant	Meaning
LVNI_ALL	Browse through all items in the index sequence.
LVNI_ABOVE	Next item above.
LVNI_BELOW	Next item below.
LVNI_TOLEFT	Return the next item on the left.
LVNI_TORIGHT	Return the next item on the right.

LVNI constants (List View Next Item) for setting the search attribute	
Constant	The returned item must ...
LVNI_FOCUSED	Have the focus.
LVNI_SELECTED	Be selected.
LVNI_CUT	Be highlighted for removal.
LVNI_DROPHILITED	Be highlighted as the potential target of a drag and drop operation.

Note how the LVM_GETSELECTEDCOUNT message is used there. With the help of this message, the macro queries the number of the selected items, which is returned as the function result of SendMessage(). In this case, no arguments are required in wParam or lParam.

```
#define ListView_GetSelectedCount(hwndLV) \
    (UINT)SendMessage((hwndLV), LVM_GETSELECTEDCOUNT, 0, 0L)
```

This information is used in the search for the selected items to permit the allocation of an array with the corresponding size on the heap, in which the number of the item is placed.

```
int *pSelectedEntries;           // Dynamic array with item number
int iSelectedCnt;                // Number of the selected items

// Query the number of the selected items
iSelectedCnt = ListView_GetSelectedCount( hListView );
if( iSelectedCnt )
{
    // Allocate array
    pSelectedEntries = malloc( iSelectedCnt * sizeof( int ) );
    if( pSelectedEntries )           // Memory still available?
    {
        int i;                       // Array-Index
        int iItem;                   // Number of the item selected

        i = 0;                       // First array item
        iItem = -1;                   // Start search at the beginning
        while( i < iSelectedCnt )    // All selected items included?
        {
            char szBuffer[ MAX_PATH ];
            // Determine the next selected item, starting from the current item
            iItem = ListView_GetNextItem( hListView, iItem, LVNI_ALL |
                                         LVNI_SELECTED );
            pSelectedEntries[ i++ ] = iItem; // Item number and take note

            ListView_GetItemText( hListView, iItem, 0,
                                szBuffer, sizeof( szBuffer ) );
            printf( szBuffer );
        }
        ...
        free( pSelectedEntries );    // Release memory
    }
}
```

Searching for items

If you're looking for a very specific item in the ListView, use the LVM_FINDITEM message or the corresponding macro ListView_FindItem(). Use the text of the item or its lParam as the search criterion.

```
#define ListView_FindItem(hwnd, iStart, plvfi) \
    (int)SendMessage((hwnd), LVM_FINDITEM, \
                     (WPARAM)(int)(iStart), \
                     (LPARAM)(const LV_FINDINFO FAR*)(plvfi))
```

The starting position of the search is again expected in wParam. However, instead of a flag, the address of a structure of the LV_FINDINFO type is required in lParam. The caller must describe the search criterion in detail in this structure before sending the message.

```
typedef struct {
    UINT    flags;                // A combination of the LVFI constants
    LPCSTR  psz;                  // Pointer to string with search text
    LPARAM  lParam;               // The search lParam
    POINT   pt;                   // Screen point as start of search
    UINT    vkDirection;          // Virtual key code for the search direction
} LV_FINDINFO;                  //
```

The creation of the structure shows that LVM_FINDITEM is apparently also used with the user control. Otherwise, it would not make sense to specify a screen coordinate in `pt` and a search direction as a key code (that is to say, arrow key) in `vkDirection`. Nevertheless, whether the control even takes these fields into consideration depends on the content of the `flags` field. A combination of the following constants must be specified here (see the following table).

LVFI constants (List View Find Information) define the search attributes.	
Constant	Meaning
LVFI_PARAM	Searches for the item with <code>lParam</code> given in <code>LV_FINDINFO.lParam</code> .
LVFI_STRING	Searches for the item with the string given in <code>LV_FINDINFO.pszText</code> .
LVFI_PARTIAL	Modifies the text search so the string given in <code>LV_FINDINFO.pszText</code> must stand at the beginning of the text in the returned item, though it can be followed by any additional characters.
LVFI_WRAP	Having arrived at the end of the list, automatically continues the search from the beginning.
LVFI_NEARESTXY	Begin the search with the item to which the screen coordinate in <code>LV_FINDITEM.pt</code> refers and search in the direction specified by the key code in <code>LV_FINDITEM.vkDirection</code> .

Depending on the flags specified here, the individual fields in `LV_FINDINFO` are either used or ignored. Specify the `LV_FINDINFO.pt` and `LV_FINDINFO.vkDirection` fields only if you also specify the `LVFI_NEARESTXY` flag in response to user input. Specify `LVFI_PARAM` to have the value in `LV_FINDINFO.lParam` considered. Then this value serves as `lParam` of the search item. The `LV_FINDINFO.psz` field, in the case of `LVFI_STRING`, must refer to the string with the text of the searched item. Also, declare the `LVFI_PARTIAL` flag for a partial search.

The result of the `SendMessage()` function tells you whether a corresponding item was found. If -1 is not returned here, you hold the index of the next suitable item in your hands. The following partial program code shows how to search for an item with a given text using this message:

```
LV_FINDINFO lfi;                // Structure for finding items
int iItem;                       // Index of the item found

lfi.flags = LVFI_STRING | LVFI_PARTIAL;    // Look for partial strings
lfi.psz = "PC Intern";                    // Text must begin with "PC Intern_"
                                           // (for example, PC Intern 6th Ed )

iItem = -1;                          // Begin search with the first item
{
    iItem = ListView_FindItem(hListView, iItem, &lfi );    // Search
    if( iItem >= 0 )
    {
        ...                                           // A hit
    }
}
while( iItem >= 0 );    // The search is continued so long as there are hits
                       // to be registered
```

Querying the incremental search string

Besides the program controlled search, the user can also look for items by inserting the initial letters of the item, while the control has the focus. The control manages an “incremental search string” for this purpose. It’s called incremental because each of the letters entered by the user is added to the string and, in the case of similar names, gets closer and closer to the searched item. A timer runs in the background, which deletes the incremental search string when it runs out, because it is assumed the user won’t make any further entries in this interaction.

You can query the current contents of the incremental search string using the LVM_GETISEARCHSTRING message. Specify the address of a buffer to receive the search string in lParam. As a precaution, reserve MAX_PATH characters for this buffer, because you don’t get an opportunity to inform the control of the size of the buffer.

```
#define ListView_GetISearchString(hwndLV, lpsz) \
    (int)SendMessage( (hwndLV), LVM_GETISEARCHSTRING, \
        0, (LPARAM)(LPTSTR)lpsz)
```

As the function result of SendMessage(), you receive the number of returned characters in the buffer. The value 0 indicates there that no incremental search is currently being carried out by the user.

Setting the visible area

The ListView control maintains a series of LVM messages. Use these messages if you only want to make sure that an item is visible or would like to execute a scrolling operation controlled by the program. It’s even possible to query the current visible area and an item’s drawing area.

Guaranteed visible

To confirm that an item is visible because, for example, it is the result of a query carried out by the user, use the LVM_ENSUREVISIBLE message or the associated macro ListView_EnsureVisible().

```
#define ListView_EnsureVisible(hwndLV, i, fPartialOK) \
    (BOOL)SendMessage( (hwndLV), LVM_ENSUREVISIBLE, \
        (LPARAM)(int)(i), MAKELPARAM((fPartialOK), 0))
```

All that is required to send the message is the number of the item, specified in wParam and the value TRUE or FALSE, specified in lParam. With lParam it is a question of whether it’s enough for the item to appear at the edge of the client area and be partly cut off by it or whether it must be visible in its full size. The latter choice is represented by FALSE.

Number of (visible) items

To determine the number of items which fit into List view or Report view in the client area of the control and are completely visible, use the LVM_GETCOUNTPERPAGE message or the associated macro ListView_GetCountPerPage().

```
#define ListView_GetCountPerPage(hwndLV) \
    (int)SendMessage( (hwndLV), LVM_GETCOUNTPERPAGE, 0, 0)
```

The message requires no arguments in wParam and lParam and returns the number of items as a function result. Getting the total number of items is just as simple. After sending LVM_GETITEMCOUNT, you get a corresponding int value as the function result. Again, no arguments are necessary in wParam or lParam.

```
#define ListView_GetItemCount(hwnd) \
    (int)SendMessage( (hwnd), LVM_GETITEMCOUNT, 0, 0L)
```

Start of the visible segment

Use the LVM_GETTOPINDEX message to query the number of the highest visible item in Report view or List view. No arguments are required in wParam or lParam.

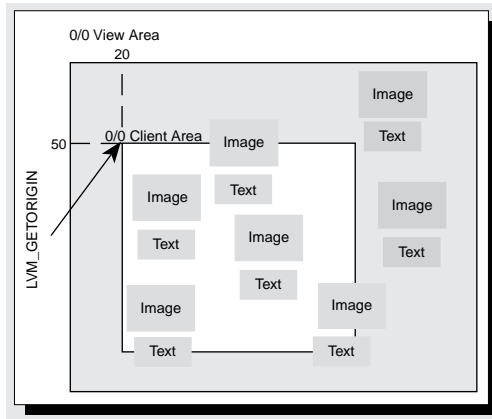

```
#define ListView_GetTopIndex(hwndLV) \
    (int)SendMessage((hwndLV), LVM_GETTOPINDEX, 0, 0)
```

As a result, you are returned either the number of the item or the value 0 if the control is not in Report or List view.

The counterpart to LVM_GETTOPINDEX is the LVM_GETORIGIN message. It can only be used in the two icon views, Small Icon view and Large Icon view. ListView_GetOrigin() is the associated macro. It expects the address of a POINT structure as its only parameter. It passes the address to SendMessage() in wParam.

```
#define ListView_GetOrigin(hwndLV, ppt) \
    (BOOL)SendMessage((hwndLV), LVM_GETORIGIN, \
        (WPARAM)0, (LPARAM)(POINT FAR*)(ppt))
```

The view area of a ListView control in the two icon views



Once the message has been successfully sent (function result = TRUE), you'll find the view coordinate in the upper-left corner of the client area, in the returned POINT structure. Because you can position the items in the two icon views at any desired location and it is not necessary for each item to follow the preceding one at a constant interval, refer to this mode as a "view area" containing all items as a sort of drawing area. It extends from coordinate 0/0 in the upper-left corner down to the lower-right corner of the last item. After sending the LVM_GETORIGIN message, you get back the coordinate of the point from the visible area currently appearing in the upper-left corner of the client area of the ListView control.

Querying the display area

In both icon views of the ListView control, you can query the bounding rectangle around all items by using the LVM_GETVIEWRECT message or the associated macro ListView_GetItemRect(). Specify the address of a RECT structure in lParam, which is to receive the View coordinates.

```
#define ListView_GetViewRect(hwnd, prc) \
    (BOOL)SendMessage((hwnd), LVM_GETVIEWRECT, \
        0, (LPARAM)(RECT FAR*)(prc))
```

The drawing area of an item

Although LVM_GETVIEWRECT returns the display area of an item, sometimes you'll also want to determine the drawing area of a specific item. You have two options. To determine only the view coordinate of the point in the upper-left corner of an item, it's best to use the LVM_GETITEMRECT message or its accompanying macro ListView_GetItemRect().

```
#define ListView_GetItemPosition(hwndLV, i, ppt) \
    (BOOL)SendMessage((hwndLV), LVM_GETITEMPOSITION, \
        (WPARAM)(int)(i), (LPARAM)(POINT FAR*)(ppt))
```

Specify the index of the item in wParam, while in lParam you specify the address of a POINT structure, in which the control places the view coordinate of the upper-left corner of the item after receiving the message. So you are determining where the item is located within the total drawing area, but not where it appears relative to the client area of the control. If you also need this information, you'll have to subtract the point that you get back when sending the LVM_GETORIGIN message from the coordinate returned here.

To determine the coordinate of the entire area covered by the item, use the `LVM_GETITEMRECT` message or its macro, `ListView_GetItemRect`. Specify the number of the item you want to query in `wParam`. Also, specify the address of a `RECT` structure in `lParam`. The control loads the View coordinates into this structure so that you get the drawing area of the item.

```
#define ListView_GetItemRect(hwnd, i, prc, code) \
    (BOOL)SendMessage( (hwnd), LVM_GETITEMRECT, \
        (WPARAM)(int)(i), \
        ((prc) ? ((RECT FAR *) (prc)) -> left = (code), \
            (LPARAM)(RECT FAR*)(prc)) \
        : (LPARAM)(RECT FAR*)NULL))
```

If you look at the macro, you'll see that another parameter named `code` is needed. It's placed in the `POINT.left` field. This code determines which part of the item the returned rectangle refers to by specifying one of the following LVIR constants. `SendMessage()` returns `TRUE` for this function if the specified item exists and is currently visible. Only in this case was the passed `RECT` structure filled with the desired view coordinates.

This partial code shows how to browse through all the items of a `ListView`, determine the bounding rectangle of its icon and check whether the item is currently visible in the client area of the control. If yes, the area covered by the image is then filled with the color black.

LVIR constants (List View Item Rect) describe the part of an item whose bounding rectangle is to be determined	
Constant	Bounding rectangle is returned around the...
LVIR_BOUNDS	Entire item, both text and image. This area in Report View also includes the subitems and the various columns.
LVIR_ICON	Icon
LVIR_LABEL	Label.
LVIR_SELECTBOUNDS	Combined icon and label but without the additional columns of Report View.

```
int iItem; // Index of the current item
POINT ptOrigin; // Origin of the ListView view
RECT rClient; // Client area of the ListView control

iItem = -1; // Begin running through the items with the first one

// Set the origin beforehand to 0, because ListView_GetOrigin returns an
// origin only in the ICON and SMALLICON view
ptOrigin.x = ptOrigin.y = 0;
ListView_GetOrigin( hListView, &ptOrigin );

GetClientRect( hListView, &rClient ); // Determines ClientArea of the
// ListView
OffsetRect( &rClient, ptOrigin.x, ptOrigin.y ); // and shift around origin
do
{
    // Get next item
    iItem = ListView_GetNextItem( hListView, iItem, LVNI_ALL );
    if( iItem >= 0 ) // items still present?
    {
        RECT rItem, rUnion;
        // Determine display zone of the item icon
        ListView_GetItemRect( hListView, iItem, &rItem, LVIR_ICON );

        // Form union from shifted client area and item rectangle
        IntersectRect( &rUnion, &rItem, &rClient );

        // Is there an overlapping rectangle?
```

```

    if( ( rUnion.right - rUnion.left > 0 ) &&
        ( rUnion.bottom - rUnion.top > 0 ) )
    {
        HDC hDC = GetDC( hListView );                // fetch ListView DC

        // Draw black rectangle in ListView DC
        HBRUSH hBR = SelectObject( hDC, GetStockObject( BLACK_BRUSH ) );
        Rectangle( hDC, rItem.left, rItem.top, rItem.right, rItem.bottom );
        SelectObject( hDC, hBR );                    // Reselect old brush
        ReleaseDC( hListView, hDC );                 //Release DC
    }
} while( iItem >= 0 );                             // Run through additional items

```

Setting the visible area

By using LVM_ENSUREVISIBLE you'll make certain that a specific item appears in the client area of the control. You cannot, however, determine whether it is visible at the upper or lower edge and, above all, which adjacent items are visible. However sometimes you'll want to have precise control over just what is presented to the user. Therefore, several options are available to, on the one hand, determine the precise position of the visible segment, and on the other hand, to set it under program control.

The LVM_SCROLL message is very important here. By using LVM_SCROLL, the visible segment can be scrolled a certain number of pixels. The potential problem is that what takes place here is a relative movement. You have to position the desired target area in relation to the currently visible area and, from that, enter the difference in pixels to be scrolled along the X- and Y-axis. Use LVM_GETITEMRECT to determine the target area in that case and LVM_GETORIGIN to determine the starting point of the currently visible area.

Sending a LVM_SCROLL message or calling the associated macro corresponds to the user moving the position indicator in the two scroll bars. The extent of the movement is described by the values specified in wParam and lParam.

```

#define ListView_Scroll(hwndLV, dx, dy) \
    (BOOL)SendMessage( (hwndLV), LVM_SCROLL, \
        (LPARAM)(int)dx, (LPARAM)(int)dy)

```

The lParam returns, for all display modes, the movement along the Y-axis (vertical scroll bar) as a value in pixels. In keeping with this, wParam determines the movement along the X-axis (horizontal scroll bar) in pixels. The only exception is in report view where the value passed in wParam is not interpreted as the number of pixels but as the number of columns by which the visible segment is to be shifted.

Querying the text width

To determine the width of the text within the ListView, use the LVM_GETSTRINGWIDTH message or the associated macro ListView_GetStringWidth. When sending the message, specify the address of the string in lParam whose width you want to determine. This is returned as the result of the SendMessage() call in pixels, the font currently set in the control being used as the basis.

```

#define ListView_GetStringWidth(hwndLV, psz) \
    (int)SendMessage( (hwndLV), LVM_GETSTRINGWIDTH, \
        0, (LPARAM)(LPCTSTR)(psz) )

```

Use the LVM_GETITEMSPACING message or its associated macro ListView_GetItemSpacing() to determine the horizontal distance between the individual items in the two icon views. You can refer to one of the two icon views, in which case the desired display must be specified in wParam of the message. TRUE represents in this case small icon view and FALSE represents the default (large) icon view.

```
#define ListView_GetItemSpacing(hwndLV, fSmall) \
    (DWORD)SendMessage((hwndLV), LVM_GETITEMSPACING, fSmall, 0L)
```

As a result, you get back the horizontal distance between the views in pixels.

Manual update

If an item has been changed, it will be displayed immediately with its new attributes, for example, with new text or a new icon. However, under certain circumstances you will force this operation, for example, because the client area of the control has been overwritten. For this purpose, use the LVM_UPDATE message or the associated macro `ListView_Update()`. The only parameter expected is the number of the item passed to the control in `wParam`.

```
#define ListView_Update(hwndLV, i) \
    (BOOL)SendMessage((hwndLV), LVM_UPDATE, (WPARAM)i, 0L)
```

If the `LVS_AUTOARRANGE` window style is available to the control, sending this message ensures that the items in both icon views will be arranged automatically.

Positioning and aligning items

In both icon views, a program can influence where the various items are displayed. First, there's the option of simply aligning the views along the border of the control or the grid and, second, of moving them to a specific place within the display area.

The `LVM_ARRANGE` message handles the alignment. All you need to do is use one of the `LVA` constants, which must be added to the message in `wParam`. The associated macro is called `ListView_Arrange()`.

```
#define ListView_Arrange(hwndLV, code) \
    (BOOL)SendMessage((hwndLV), LVM_ARRANGE, (WPARAM)(UINT)(code), 0L)
```

Specify the desired arrangement of the images using `LVA` constants in `wParam` (see table to the right).

Manual positioning

Two messages, `LVM_SETITEMPOSITION` and `LVM_SETITEMPOSITION32`, are available for positioning individual items manually. Both accept the number of the item to be positioned in `wParam` and the coordinate in `lParam`. However, the messages require different forms, because, while `LVM_SETITEMPOSITION` works with 16-bit coordinates, `LVM_SETITEMPOSITION32` uses 32-bit coordinates. That's why the messages also have different names. The coordinates refer to the display area. This is the pixel area in which the various items of the control appear. Its point of origin is the coordinate (0/0), which determines the upper-left corner of the viewing area and first appears in the upper-left corner of the client area.

The size of the viewing area is restricted only by the specified coordinates. If an item is dragged past the border of the previous viewing area, the latter will be correspondingly enlarged. When using `LVM_SETITEMPOSITION32`, the viewing area can therefore become much larger, after all, you are working with 32-bit coordinates. However, this is the only difference, which is why you can usually restrict yourself to working with `LVM_SETITEMPOSITION` and the associated macro, `ListView_SetItemPosition()`. In this case, the two 16-bit wide X-coordinates and Y-coordinates are combined to form the `lParam` of the message using `MAKELPARAM`.

```
#define ListView_SetItemPosition(hwndLV, i, x, y) \
    (BOOL)SendMessage((hwndLV), LVM_SETITEMPOSITION, \
        (WPARAM)(int)(i), MAKELPARAM((x), (y)))
```

LVA constants (List View Arrange) determine the arrangement of the images	
Constant	Meaning
<code>LVA_DEFAULT</code>	Alignment as specified in the window style (<code>LVS_ALIGNLEFT</code> or <code>LVS_ALIGN_TOP</code>).
<code>LVA_ALIGNLEFT</code>	Align images along left border of the window.
<code>LVA_ALIGNTOP</code>	Align images along top border of the window.
<code>LVA_SNAPTOGRID</code>	Leave images as they are but snap to grid.

It's different with LVM_SETITEMPOSITION32 and the associated macro `ListView_SetItemPosition32()`. Since both coordinates are of the long type, they cannot be combined into an `LPARAM`. These items are referenced for this reason using a structure of the `POINT` type, whose address must be specified in the `LPARAM` of the message. You don't need to worry about this problem when you use the macro. It creates a new code block with a local `POINT` variable named `ptNewPos`. The passed X-coordinate and Y-coordinate are loaded into this variable before they're sent using the `SendMessage()` call.

```
#define ListView_SetItemPosition32(hwndLV, i, x, y) \
{ POINT ptNewPos = {x,y}; \
  SendMessage(hwndLV, LVM_SETITEMPOSITION32, \
    (LPARAM)&ptNewPos, (LPARAM)i); \
}
```

Drawing items

To have the control redraw certain items, use the message `LVM_REDRAWITEMS` or its associated macro `ListView_RedrawItems()`. Specify the index of the first item to be redrawn in `wParam`, and specify the index of the last such item in the `LPARAM`. The items must be currently visible in the client area of the control for a successful result.

```
#define ListView_RedrawItems(hwndLV, iFirst, iLast) \
(BOOL)SendMessage(hwndLV, LVM_REDRAWITEMS, \
  (LPARAM)iFirst, (LPARAM)iLast)
```

This message causes the control merely to note the named items for redrawing. They're drawn only after the control has received a `WM_PAINT` message. For example, you can generate this message using the API function `UpdateWindow()`. Do not use this message to change the items using the various LVM messages, for example, `LVM_SETITEM` or `LVM_SETITEMPOSITION`. The control automatically makes certain the changed items are redrawn if they appear in the client area of the control.

Setting texts and images using callbacks

The `ListView` control usually receives the text and the images of the items from the various `LV_ITEM` structures as they are specified when an item is inserted. However, the control can also be induced to query this information dynamically each time an item is displayed. In this case, it sends an `LVN_DISPINFO` message to the parent to make it aware of the item's text and/or image. It's also possible in this way to effect a sort of owner draw. That makes sense, for example, when a program likewise maintains the image and text of an item in its own data structures and it would therefore be inefficient to store it a second time with the item itself. This requires several items; otherwise, the additional work would outweigh the amount of memory saved.

All that is necessary to start this mechanism is to load two special constants during the insertion or setting up of an item (`LVM_INSERTITEM` or `LVM_SETITEM`) for the text and/or image. If the text of the item is to be queried with each display, specify the `LPSTR_TEXTCALLBACK` constant in the `LV_ITEM.pszText` field. It corresponds to the address `0xFFFFFFFF` and therefore to the value `-1`. Handle the `LV_ITEM.iImage` field in exactly the same way. The `L_IMAGECALLBACK` constant is used in this case if the number of the image is to be queried with each display. This constant also corresponds to the value `-1`.

Querying using LVN_GETDISPINFO

When it is time to display an item validated in this way, the control will send a `WM_NOTIFY` message to its parent window. You will find the address of an `LV_DISPINFO` structure in the `LPARAM` of this message. Located at the beginning of this structure, as usual, is a variable of the `NMHDR` type, by which the message is first identified as `LVN_GETDISPINFO`.

```
typedef struct {
    NMHDR    hdr;                // Notification message header
    LV_ITEM  item;               // ListView item
} LV_DISPINFO;
```

The second field, named `item`, contains an `LV_ITEM` structure describing the item to be queried and offering space for transfer of the required information. You can determine which item it is from the index in the `LV_ITEM.iItem` field or from the assigned `lParam` in the `LV_ITEM.lParam` field. The `LV_ITEM.mask` field shows which information is being queried. If the text is requested, set the `LVIF_TEXT` flag in the field, if the image is queried, set the `LVIF_IMAGE` flag. It is also possible for two flags to be active at the same time, if both attributes are to be queried. Place the text in the `LV_ITEM.pszText` field, place the number of the image as an index in the image list set in the `LV_ITEM.iImage` field.

If the specified image or text is to remain constant and thus, does not need to be queried further, simply set the `LVIF_DI_SETITEM` flag in the `LV_ITEM.mask` field. The control then notes the specified information and does not send any additional `LVN_GETDISPINFO` messages relative to the item.

Displaying a text change

Problems can occur if the text of an item must be queried by the control using an `LVN_GETDISPINFO` message and the user edits the text. The parent must somehow be informed of this. Otherwise, it will not note it and in the next query, will return the old text instead. The control therefore sends its parent an `LVN_SETDISPINFO` message in this situation, whose `lParam` points to a structure of the `LV_DISPINFO` type, as in the case of `LVN_GETDISPINFO`. In turn, this structure contains an `LV_ITEM` structure that identifies the item. You will find the pointer to a buffer with the new text in this structure's `pszText` field. Take note of this pointer and return it for the item the next time an `LVN_GETDISPINFO` message comes.

Editing text

If the `LVS_EDITLABELS` flag is set in the window style of a `ListView` control, the control gives the user the option of editing the text of an item after clicking it. However, a program can also trigger this process automatically by using the `LVM_EDITLABEL` message or the associated macro, `ListView_EditLabel()`. The number of the item to be edited must be specified in `wParam`. As a result, the window handle of the edit control or the value `NULL` (if the editing could not be set into motion) is returned.

```
#define ListView_EditLabel(hwndLV, i) \
    (HWND)SendMessage(hwndLV, LVM_EDITLABEL, (WPARAM)(int)(i), 0L)
```

To discontinue the current editing, send the message again, specifying the value `-1` in `wParam` instead of the number of the item. Note that the call can only be executed successfully if the `ListView` control has the focus. If this is not the case, first use the API function, `SetFocus()` to send the focus to the control. If the item of interest is not visible when the `LVN_EDITLABEL` message is sent, the control will scroll the visible section. This enables the item to appear and be edited.

Suppressing the input

Regardless of whether the editing of the text is set into motion by the user or by the program: Before editing can begin, the parent receives an `LVN_BEGINLABELEDIT` message in the context of a `WM_NOTIFY` message, whose `lParam` points to a structure of the `LV_DISPINFO` type. It contains initially the usual structure of the `NMHDR` type, which identifies the message and its sender. A structure of the `LV_ITEM` type then follows, in whose `lParam` and `iItem` fields the item whose text is to be edited is validated.

```
typedef struct {
    NMHDR    hdr; // Notification message header
    LV_ITEM  item; // ListView item
} LV_DISPINFO;
```

The parent now has the option of suppressing editing by returning `TRUE` in its message procedure. `FALSE`, which is automatically returned when the message is not being processed, accepts the editing.

Concluding the input

The parent, on completing the input, likewise receives a notification from a `WM_NOTIFY` message, called an `LVN_ENDLABELEDIT` message. The `lParam`, points to an `LV_DISPINFO` message whose `item` field contains an

LV_ITEM structure with a new text in the pszText field. A sign the user has cancelled the input with Escape is if a NULL pointer is found there. If so, no new text was entered. In that case, call LV_SETITEM with the new text if the latter is to be retained permanently because the control does not automatically anchor the new text in the LV_ITEM structure.

Access to the edit control during editing

To start the edit control in which the user is editing the text, determine the handle by using the LVM_GETEDITCONTROL message or its associated macro ListView_GetEditControl(). No parameters are expected in wParam or in lParam, although the handle as the result of the SendMessage() call is returned. The function result will be NULL if no item is currently being edited.

```
#define ListView_GetEditControl(hwndLV) \
    (HWND)SendMessage( (hwndLV), LVM_GETEDITCONTROL, 0, 0L)
```

Subclassing the edit controls

A small problem arises if the ListView control is part of a dialog box and the user presses ENTER or ESCAPE when editing the text to terminate the input. The corresponding WM_KEYDOWN messages are immediately sent to the message procedure of the dialog box. The message procedure interprets these messages as a request to press the default buttons for these keys. (As a rule, OK for ENTER and Cancel for ESCAPE.) The dialog box then closes, although the user really only wanted to end the editing of the text in the ListView control.

To prevent this, use the edit control to perform what is called subclassing (see the previous section of this Chapter kkk with the example of a TreeView control).

Working in report view

More is involved in report view than setting up icon lists and inserting a few items. To display subitems, you first must create their columns. It's possible to specify a series of column attributes like column width, the column header and the alignment of the subitems in the column. Finally, the individual subitems are loaded, in other words, the text that is to appear in the various columns of a line.

Whatever appears in the first column is always the actual text of the item, as inserted in the LV_ITEM.pszText field using LVM_INSERTITEM. The small icon of the item appears on the left if an Image List has been loaded. The individual subitems can be neither selected nor edited.

The LV_ITEM.iSubItem field, scarcely noticed in the other three display modes, is very important and always initialized with 0. This zero represents "main item", whereas each value, starting with one, is the number of a subitem. To apply one of the various LVM messages to a subitem, little is changed compared to the previous procedure. Simply make certain the proper subitem number appears in the LV_ITEM.iSubItem field. The number of columns set up in that case determines how many subitems per entry can be inserted. The valid subitem index ranges from 1 to n.

The control does not have to be in report view to allow the setup of columns and subitems. However, these operations don't become visible until you switch the control to report view using its window style.

The construction of a report ListView at runtime

Once the control is generated using CreateWindowEx(), create the main items next. Otherwise, it's not possible to insert any subitems. For one, it is necessary, when sending the LVM_INSERTITEM message, to insert an associated main item in the LV_ITEM.iItem field, to which the subitem is to be assigned. The columns can be set up when the main items are generated.

The corresponding message is called LVM_INSERTCOLUMN and the associated macro called ListView_InsertColumn(). Specify the number of the new column in wParam when sending the message. Specify the address of a structure of the LV_COLUMN type in lParam to describe the attributes of the new column. The specified column number is returned with SendMessage() if the column can be inserted as desired. A -1 is returned in case of an error.


```
#define ListView_InsertColumn(hwnd, iCol, pcol) \
    (int)SendMessage((hwnd), LVM_INSERTCOLUMN, \
        (WPARAM)(int)(iCol), \
        (LPARAM)(const LV_COLUMN FAR*)(pcol))
```

The LV_COLUMN structure contains fields for all attributes of a column. This includes the width, the text of the column heading, the alignment of the columns or the index of the subitems to be displayed in this column.

```
typedef struct {
    UINT    mask;                // Combination of the LVCF flags
    int     fmt;                // Alignment of the columns, one of the LVCFMT constants
    int     cx;                 // Width of the column in pixels
    LPSTR   pszText;            // Pointer to buffer with column header
    int     cchTextMax;         // Size of the buffer, points to the pszText
    int     iSubItem;           // Number of the subitems appearing in the column
} LV_COLUMN;                  //
```

Use the mask field to indicate which fields are initialized. This is achieved with a combination of the LVCF constants listed in the table to upper right.

Use the LV_COLUMN.fmt field to determine the alignment of the text in the column. Note that this doesn't affect the alignment of the column heading because it's always centered. One of the LVCFMT constants listed in the table to lower right must be specified.

Use the LV_COLUMN.cx field to set the width of the column in pixels. LV_COLUMN.pszText must point to the string with the column header (if a column heading is used). The LVS_NOCOLUMNHEADER flag must not be set in the window style of the control. Setting this flag suppresses all column headers. Also, do not initialize LV_COLUMN.cchTextMax when you insert a new column. It's only needed if the LV_COLUMN structure is used with the LVM_GETCOLUMN message to determine the attributes of a column. More about that below.

LVCF constants (ListView Column Flags) show the initialized fields in an LV_COLUMN structure

Constant	Which field is initialized ...
LVCF_FMT	The fmt field
LVCF_WIDTH	The cx field
LVCF_TEXT	The pszText field
LVCF_SUBITEM	The iSubItem field

LVCFMT constants (List View Column Format) determine the alignment of the text in a column

Constant	Alignment of the text
LVCFMT_LEFT	Left justified
LVCFMT_RIGHT	Right justified
LVCFMT_CENTER	Centered
LVCFMT_JUSTIFYMASK	Adds the three above flags

Finally, set the number of the subitem which is to appear in the column using the LV_COLUMN.iSubItem field. You are not forced to set all the same subitems in the same sequence as they are to appear in the individual columns of the control, but can select the sequence yourself and also even change it at a later time.

Setting the column attributes

Use either the LVM_SETCOLUMN message or the associated macro, ListView_SetColumn() for this purpose. Again, wParam requires the number of the column addressed and lParam requires the address of an LV_COLUMN structure.

```
#define ListView_SetColumn(hwnd, iCol, pcol) \
    (BOOL)SendMessage((hwnd), LVM_SETCOLUMN, \
        (WPARAM)(int)(iCol), \
        (LPARAM)(const LV_COLUMN FAR*)(pcol))
```

To determine which attributes of the indicated column are to be changed, load the corresponding LVCF constants in the LV_COLUMN.mask field and initialize the associated fields in LV_COLUMN, such as pszText or cx. If the changes could not be made as desired, TRUE is returned as the result of SendMessage(); otherwise FALSE is returned.

Querying the column attributes

In the opposite direction, you can query the attributes of a column using the LVM_GETCOLUMN message of the associated macro `ListView_GetColumn()`. Pass the number of the desired column in `wParam`, and pass the address of an `LV_COLUMN` structure which is to receive the various attributes in `lParam`.

```
#define ListView_GetColumn(hwnd, iCol, pcol) \
    (BOOL)SendMessage((hwnd), LVM_GETCOLUMN, \
        (WPARAM)(int)(iCol), (LPARAM)(LV_COLUMN FAR*)(pcol))
```

Use the mask field to show which attributes are queried. If the column header is to be queried, the `LV_COLUMN.pszText` must be initialized before sending the message with the address of a buffer, which is to accept the column header. Use the `LV_COLUMN.cchTextMax` field to specify its size (in characters) to the control. This prevents the buffer from being accidentally overwritten with a longer text. TRUE is returned as a function result if the specified column could not be queried as desired; otherwise, FALSE is returned.

Access to the column width

Use two special messages to simply set or query the width of a column. These messages do not need an `LV_COLUMN` structure. `LVM_SETCOLUMNWIDTH` and the associated macro `ListView_SetColumnWidth()` expect the number of the addressed column in `wParam` and the new width in pixels in `lParam`.

```
#define ListView_SetColumnWidth(hwnd, iCol, cx) \
    (BOOL)SendMessage((hwnd), LVM_SETCOLUMNWIDTH, \
        (WPARAM)(int)(iCol), MAKELPARAM((cx), 0))
```

Use a similar technique with the `LVM_GETCOLUMNWIDTH` message or the associated macro `ListView_GetColumnWidth()`. When sending the message, specify the number of the addressed column in `wParam`. You get back the width of the column in pixels as the function return of `SendMessage()`.

```
#define ListView_GetColumnWidth(hwnd, iCol) \
    (int)SendMessage((hwnd), LVM_GETCOLUMNWIDTH, \
        (WPARAM)(int)(iCol), 0)
```

Deleting a column

Use the `LVM_DELETECOLUMN` message or the associated `ListView_DeleteColumn()` macro to delete a column. You do not need to add an argument next to the column number in `wParam`.

```
#define ListView_DeleteColumn(hwnd, iCol) \
    (BOOL)SendMessage((hwnd), LVM_DELETECOLUMN, \
        (WPARAM)(int)(iCol), 0)
```

The function result lets you know whether the column could be successfully deleted. In this case, the function result is TRUE.

Sorting with column headers

If you have worked with the Windows Explorer, you are aware of the option of sorting the listing of files in report view by clicking the column headers. You can have the sequence determined by the alphabetical listing of the file names, or by the size of the file. Clicking the corresponding column header is enough. Another click reverses the sorting sequence. Since this is so convenient for users, it might be a good idea to add this functionality to your dialog boxes.

Although the `ListView` control supports the implementation of such a sorting functionality, it doesn't do all your work. First, the parent procedure must respond in its message procedure to the `LVN_COLUMNCLICK` message. It receives this message in the context of a `WM_NOTIFY` message from the `ListView` control. The message indicates the user has clicked the head of a column with the column header. Do not explicitly set the `LVS_NOSORTHEADER` flag in the window style of the control. Setting this flag prevents the control from registering the clicking of a column header.

When the WM_NOTIFY message is received, lParam points to a structure of the NM_LISTVIEW type. This type, on the one hand, identifies the LVN_COLUMNCLICK message and, on the other, returns information on the column clicked. However, most fields are not initialized.

```
typedef struct {
    NMHDR    hdr;                // The usual header of an WM_NOTIFY message
    int      iItem;              // -1 (for all)
    int      iSubItem;           // Number of the SubItem whose column was clicked
    UINT     uNewState;          // 0
    UINT     uOldState;          // 0
    UINT     uChanged;           // 0
    POINT    ptAction;           // 0
    LPARAM   lParam;             // 0
} NM_LISTVIEW;
```

You get the number of the subitem whose column was clicked from the iSubItem field. This subitem must therefore be retained as the sorting criterion. However you must trigger the sort yourself by accessing the LVM_SORTITEMS message. It will do most of the work but does demand a sorting function from the caller so the items are put in the correct order. The following section explains how this is done.

Sorting items

To sort items based on a self-defined sorting criterion, use the LVM_SORTITEMS message or the associated macro ListView_SortItems(). The sort is developed using a function of the caller, which is invoked by the control for comparing two items. The address of this function, which must be of the PFNLVCOMPARE type, has to accompany the message in lParam. Specify any value in wParam, which is passed to the sorting function with each call and offers the option of creating a link between the trigger of a sorting operation and the sorting function.

```
#define ListView_SortItems(hwndLV, _pfnCompare, _lPrm) \
    (BOOL)SendMessage( (hwndLV), LVM_SORTITEMS, \
        (WPARAM)(LPARAM)_lPrm, \
        (LPARAM)(PFNLVCOMPARE)_pfnCompare)
typedef int (CALLBACK *PFNLVCOMPARE)(LPARAM item1,
                                     LPARAM item2, LPARAM lPrm);
```

The sorting function receives the value from the LV_ITEM.lParam field in the first two parameters, as specified when the item was inserted using LVM_INSERTITEM. You receive the value specified in wParam when the LVM_SORTITEMS message was sent in the third parameter. The sorting function must decide based on the first two parameters which of the two items attached to it is to appear first. This decision is reported to the control through the function result. A value of 0 indicates that the item named first is to appear before the second, a value greater than 0 implements the reverse sequence. The returned value is 0 if both entries are identical relative to their sorting sequence.

Note that the numbers of the items are necessarily changed by sorting. If you store the numbers of specific items in your program, you must therefore determine these again, for example, using LVM_FINDITEM.

Responding to mouse and keyboard events

To respond to the clicking of an item with the mouse, use a corresponding message. As with ListView notifications (the LVN messages), the control here must also use a WM_NOTIFY message which it sends to the parent of the control. In this case, the lParam points to an NMHDR structure from which it is possible to determine the author of the message (hwndFrom field) and the actual message code (code field).

```
typedef struct {
    HWND    hwndFrom;           // Window handle of the sender
    UINT    idFrom;             // Control ID of the sender
```

```

        UINT    code;                                // The message coded
    } NMHDR;

```

Because many controls use this option, no special LVN messages are returned with mouse events. Instead, NV messages are returned (see the table below). They also indicate to the parent the transmission and the loss of the focus by the control.

NM messages (Notification Messages) describe interaction events	
Message	Event
NM_CLICK	A mouse click on the client area with the primary (left) mouse button.
NM_DBLCLICK	A double-click on the client area with the primary (left) mouse button.
NM_RCLICK	A mouse click on the client area with the secondary (right) mouse button.
NM_RDBLCLICK	A double-click on the client area with the secondary (right) mouse button.
NM_RETURN	The user pressed the Enter key when the control had the focus.
NM_SETFOCUS	The control obtained the focus.
NM_KILLFOCUS	The control lost the focus.

You may need more information besides the returned NMHDR structure. This is particularly true with mouse events and specifically the mouse position and the item clicked. However, you must put this information together yourself by first loading the mouse position using the API function, `GetCursorPos()`, into a `POINT` structure specified when the call was made. What you then get back are screen coordinates which have to be set according to the control - in this case the `ListView` control.

In the second step, call the API function `ScreenToClient()` with the returned `POINT` structure and the handle of the `ListView` control. That recomputes the coordinates relative to the upper-left corner of the client area of the control. Both functions are likewise contained in the `USER32.DLL` and defined in the Include file `WINUSER.H`.

Although you now have the necessary coordinates, you'll surely want to assign them to a specific item to be able to respond to the selection of the user. The `ListView` control uses a message named `LVM_HITTEST` for this purpose. This message can also be accessed through the associated macro `ListView_HitTest()`. In that case a structure of the `LV_HITTESTINFO` type must be passed in `lParam`.

```

define ListView_HitTest(hwndLV, pinfo) \
    (int)SendMessage((hwndLV), LVM_HITTEST, \
        0, (LPARAM)(LV_HITTESTINFO FAR*)(pinfo))

```

Before the call, place the coordinate of the point in the `pt` field of the `POINT` structure. To make it really simple, merely specify the address of the `POINT` structure from the `LV_HITTESTINFO` variable during the prior calls to `GetCursorPos()` and `ScreenToClient()`. Then you won't have to put up with the inconvenience of loading the coordinate later into the `LV_HITTESTINFO` structure.

```

typedef struct {
    POINT pt;                                // Coordinates of the point
    UINT  flags;                            // Accepts mail, a combination of the LVHT constants
    int   iItem;                            // The control puts the index of the item here
} LV_HITTESTINFO;

```

If a value not equal to -1 is returned by `SendMessage()`, it means it was possible to assign the specified coordinate to an item in the control. The return value corresponds in this case to the index of the item which is also included in the `iItem` field of the `LV_HITTESTINFO` structure.

Even if the specified screen point could not be assigned to the item, you'll find, in the `LV_HITTESTINFO`.`flags` field, one or more of the `LVHT`

LVHT constants (List View Hit Test) describe the position of a given point relative to a ListView and its items.	
Constant	The specified point is located ...
LVHT_ABOVE	Above the client area of the control.
LVHT_BELOW	Below the client area of the control.
LVHT_TORIGHT	To the right, next to the client area of the control.
LVHT_TOLEFT	To the left, next to the client area of the control.
LVHT_NOWHERE	Within the client area, but not above one of the various items.
LVHT_ONITEMICON	Above the icon of an item.
LVHT_ONITEMLABEL	Above the text of an item.
LVHT_ONITEMSTATEICON	Above the state icon of an item.

This program code shows how the parent of a ListView control responds in its message procedure to the receipt of an NM_DBLCLICK message, first determining the addressed item and then its text. The text is displayed using MessageBox().

```

h( LOWORD( wp ) )          // Which control is sending notification?
{
    case IDC_LISTVIEW:      // ListView control
    {
        LPNM_LISTVIEW pnmlv = (LPNM_LISTVIEW)lp;
        switch( pnmlv->hdr.code)    // Evaluate notification code
        {
            case NM_DBLCLK:        // Double click
            {
                LV_HITTESTINFO hi; // For determination of the item under pointer
                int iItem;          // Item found

                // Determine mouse position (screen coordinates)
                GetCursorPos( &hi.pt );
                // Convert into ListView client coordinates
                ScreenToClient( hListView, &hi.pt );

                // Which item is below the cursor?
                iItem = ListView_HitTest(hListView, &hi );

                if( iItem >= 0 )    // Item found?
                {
                    char szBuffer[ MAX_PATH ];          // Text buffer
                    ListView_GetItemText(hListView,    // Determine text of item
                                         iItem,
                                         0,
                                         szBuffer,
                                         sizeof( szBuffer ) )
                    MessageBox( hWnd, szBuffer, "Item text", 0 );    // and display
                }
            }
            break;
        }
    }
}
break;

```

<! Note to Paul - The first line of the program code did not appear in the translation - I chanced upon it in the book & added it myself. Al Be Ware !>

Responding to keyboard events

Besides the NM_RETURN message informing the parent the Enter key was pressed, the parent receives the code of the pressed key from a different message, LVN_KEYDOWN, which likewise sails under the flag of a WM_NOTIFY message. You get this message when the ListView control has the focus and the user presses any key. It doesn't matter whether the key causes an action (for example, selecting a different item). You receive the address of an LV_KEYDOWN structure in lParam containing more detailed information on the event.

```
typedef struct {
    NMHDR  hdr;    // Ordinary NMHDR structure with info on sender and message
    WORD   wVKey;  // Virtual key code for the key pressed
    UINT   flags;  // 0
} LV_KEYDOWN;
```

Besides the usual NMHDR structure whose code field contains the code of LVN_KEYDOWN, the virtual key code of the pressed key is specified in the wVKey field. The flags field is not being used at present and is always initialized with 0.

Drag and Drop

Drag and Drop support is a key component of modern applications. The ListView control also supports this technology, which provides the essential foundation for letting users move items within the various views. However, the control does not handle this process by itself; there is only a certain degree of support.

Drag and Drop operations are a relatively complicated business involving several operations. A simple function call or the sending of an LVM message unfortunately won't take care of it, even if it is supported to some extent by the control. You have to add a relatively large amount of your own code, which is mainly used in the message procedure of the parent window. Our example uses the message procedure of the dialog box for the LVDEMO program.

The start of a drag operation

The entire process begins here when the parent receives a WM_NOTIFY message from its ListView control. On closer inspection this message turns out to be an LVN_BEGINDRAG or LVN_BEGINRDRAG notification. The first message indicates the beginning of a drag operation using the primary (left) mouse button (LVN_BEGINDRAG). The second message concerns a drag operation using the right mouse button (LVN_BEGINRDRAG). In both cases, a pointer to an NM_LISTVIEW structure is found in lParam of the message.

```
typedef struct {
    NMHDR  hdr;    // The usual header of a WM_NOTIFY message
    int     iItem;  // Number of the dragged item
    int     iSubItem; // 0
    UINT    uNewState; // 0
    UINT    uOldState; // 0
    UINT    uChanged;  // 0
    POINT   ptAction;  // Mouse pointer position at the beginning of the
                      // drag operation
    LPARAM  lParam;    // 0
} NM_LISTVIEW;
```

Besides the hdr field, only the iItem field with the number of the dragged ListView item and the ptAction field with the coordinate of the mouse cursor at the beginning of the Drag operation are initialized in this structure. We'll need this coordinate soon.

In the next step, the first job is to build an image which, in the context of the drag operation, is dragged across the screen with the mouse cursor. The ListView control makes a special message available for this purpose, named LVM_CREATEDRAGIMAGE, which can also be called with the associated macro, ListView_CreateDragImage().

```
#define ListView_CreateDragImage(hwnd, i, lpptUpLeft) \
    (HIMAGELIST)SendMessage((hwnd), LVM_CREATEDRAGIMAGE, \
        (WPARAM)(int)(i), \
        (LPARAM)(LPPOINT)(lpptUpLeft))
```

When the message is sent, wParam contains the number of the item as taken from the NM_LISTVIEW structure. Also, lParam contains the address of a POINT structure. This is where the control deposits the view coordinate, where the drag image generated by the call is initially drawn. This is a job for your program; more on that later.

When you send an LVM_CREATEDRAGIMAGE message, you receive the handle of an image list that was generated by the control for the execution of the drag operation as the function result of the SendMessage() function. The image list contains only a single image. It includes the image of an item and its text. It must always follow the mouse pointer during the drag operation as a *drag image*.

Creating the drag image

Before getting started, you need to set this image as a drag image using ImageList_BeginDrag() function. The function expects the handle of the image list, the index of the image (here always 0) and the distance between the hotspot of the cursor and the upper-left corner of the image.

```
BOOL ImageList_BeginDrag(           // Starts dragging with an image from the IL
    HIMAGELIST himl,                // Handle of the image list
    int iTrack,                      // Number of the image
    int dxHotspot,                   // X-distance of the drag point from the upper
                                     // lefthand image corner
    int dyHotspot                    // Ditto Y (both entries in pixels)
);
```

The two hotspot entries refer to the distance between the point where the hotspot of the mouse cursor was at the beginning of the drag operation and the upper-left corner of the image. This distance should remain constant as the image is dragged across the screen. This prevents the user from getting the impression that the image is floating beneath the mouse cursor. Instead, the image should always move parallel to the mouse cursor. That is why you have to set this distance when calling ImageList_BeginDrag().

Calculate the distance by subtracting the X-coordinate returned by ListView_CreateDragImage() from the X-coordinate returned by NM_LISTVIEW.ptAction. Do the same with the Y-coordinate to determine the required distance.

Disabling scrolling

Next, use SetWindowLong() to manipulate the window style of the ListView control. This requires enabling the LVS_NOSCROLL flag to prevent the visible area in the ListView from being scrolled during the drag operation. While this is slightly impractical, because the drag target remains restricted to the visible segment, it's the only way this can be done (not even with Windows Explorer). Otherwise you could run into confusion with the representation of the drag operation and the items lying below them. At the end of the drag operation, turn the LVS_NOSCROLL window style off again. However, we're not to that point yet.

In the next step, the current pointer position is determined with the API function, GetCursorPos(). The returned coordinate is passed on to the Image List function, ImageList_DragEnter(), which at the same time receives the handle of a window. This handle defines the window to which the drag operation will remain limited. The drag image is cut off at the borders of the specified window and no longer visible outside the window.

```
BOOL ImageList_DragEnter(
    HWND hwndLock,                  // Window to which the drag operation remains
                                     // restricted
    int x,                          // X-coordinate for the start of the drag-image display in
                                     // the window
```

```

    int    y          // Y-coordinate for the start of the drag-image display in
                      // the window
};

```

If you specify NULL as the `hwndLock` parameter, the entire desktop will be considered as the window for the drag operation. Also, the specified coordinates will be interpreted as the distance from the upper-left corner of the screen. `ImageList_DragEnter()` also immediately draws the drag image at the indicated position relative to the upper-left corner of the specified window, so the reaction to the `LVN_BEGINDRAG` message is essentially finished.

One final task remains. Set the capture to the parent window of the `ListView` controls (in this case, the dialog box) using the API function `SetCapture()`, so that it will continue to receive all mouse messages. This ensures that the user can guide the mouse pointer over the screen without having any of the controls it “runs over” receiving mouse messages. After all, the other controls on the screen are supposed to remain in place during the drag operation. After the `SetCapture()` call, all mouse messages will go directly to the message procedure of the dialog box, even if the mouse cursor is outside this window.

The following code was taken from our sample program, `LVDEMO`, and shows the steps we’ve been discussing. This program appears at the end of the chapter. You see the beginning of the dialog box message procedure, where after receiving a `WM_NOTIFY` message, the procedure responds to the `LVN_BEGINDRAG` message. Besides the actions described, an internal drag flag called `bDragging` is also set here. This is done so the message procedure will still know the next time it receives a message that it is in a drag context. Also, the number of the dragged item is stored in the `iDragItem` variable so you can access it later if it is dropped somewhere.

```

BOOL WINAPI DlgProc( HWND hWnd, UINT wMsg, WPARAM wp, LPARAM lp )
{
    // Status variables for simplified access to control elements
    static HWND      hListView;          // ListView handle

    static BOOL      bDragging;          // Drag process?
    static int       iDragItem;          // Index of the item to be dragged
    static int       iDropTarget;        // Index of the DropTarget
    static HIMAGELIST hDragImage;        // Drag image list
    static POINT     ptHotSpot;

    switch( wMsg )
    {
        case WM_NOTIFY:
            switch( LOWORD( wp ) )
            {
                case IDC_LISTVIEW:
                {
                    LPNM_LISTVIEW pnm_lv = (LPNM_LISTVIEW)lp;
                    switch( pnm_lv->hdr.code )
                    {
                        case LVN_BEGINDRAG:          // Begin drag process
                        {
                            POINT pt,          // For screen coordinates of the mouse
                                ptItem;        // Coordinates of the drag point

                            // Determine drag image —————
                            hDragImage = ListView_CreateDragImage( hListView,
                                                                    pnm_lv->iItem,
                                                                    &ptItem );

                            // Compute position of the mouse cursor in the DragImage —

```

```

    ptHotSpot.x = pnmlv->ptAction.x - ptItem.x;
    ptHotSpot.y = pnmlv->ptAction.y - ptItem.y;

    // Prepare image list for dragging -----
    if (!ImageList_BeginDrag(hDragImage,
                            0,
                            ptHotSpot.x,
                            ptHotSpot.y ) )

        return FALSE;

    // No movements in the ListView during the drag -----
    SetWindowLong() hListView,
        GWL_STYLE,
        GetWindowLong( hListView, GWL_STYLE) |
        LVS_NOSCROLL );

    GetCursorPos( &pt );
    ImageList_DragEnter(NULL, pt.x, pt.y );           // Start drag

    SetCapture( hWnd );                             // Set capture to dialog
    bDragging = TRUE;
    iDropTarget = -1;                                // So far no drop target
    iDragItem = pnmlv->iItem;                          // Note drag item
}
break;
}
break;
}
break;

```

Trailing the drag image

The user remains in drag mode if the mouse button is not released and the message procedure doesn't receive a WM_LBUTTONDOWN from the WM_RBUTTONDOWN message. The drag item must, therefore, follow the mouse cursor with each mouse movement and corresponding receipt of a WM_MOUSEMOVE message.

Furthermore, you may also want to validate the screen controls lying below the mouse cursor as potential drop targets according to the logic of the drop operation. This requires additional program code to identify the screen control. This determines whether it qualifies as the target (and therefore as a drop location) of the drop operation. The Explorer consumes a considerable of resources for this, dragging folders and files to the Desktop and dropping them there. This involves, under certain circumstance, extensive copying operations in the files system.

In our case, we want to stick with targets within the current ListView, so the dragged item can be dropped only over another item or in the empty space between items. While the item is being dragged over another one, the latter should be validated as a potential drop site. ListView entries with the constant LVS_DROPHILITED have a special style for this purpose. Use the state field of an LV_ITEM entry (using LVM_SETITEM) to switch LVS_DROPHILITED on/off.

If it were only a matter of having the drag image follow the mouse cursor, it would be quite simple, for the Image List function, ImageList_DragMove(), combines all the functions needed to do that. It first makes the drag image transparent at its old position and again restores the old screen content, before drawing the drag image at its new position. Once a drag operation is begun by calling ImageList_BeginDrag(), you need only call ImageList_DragMove() again, each time a WM_MOUSEMOVE message is received, to cause the drag image to follow. As your argument, specify the two coordinates obtained in lParam in the context of the WM_MOUSEMOVE message. These must be converted in advance for the window specified in the

ImageList_DragEnter(), which means in this case: relative to the upper-left corner of the screen. With the aid of the API function, ClientToScreen, this conversion, with the specification of the window handle of the dialog box and the coordinates, is no problem.

Here is the ImageList_DragMove() function which receives the coordinates as parameters.

```

BOOL ImageList_DragMove(
    int x,                // New X-coordinate (relative to Desktop)
    int y                // New Y-coordinate (relative to Desktop)
);

```

Problems may occur if the screen controls under the drag image have been validated as the target of the drop operation. If the user moves the mouse cursor further, he will under certain circumstances leave the zone of influence of the item currently highlighted, which must therefore be redrawn in its original state. Furthermore, the mouse cursor is now possibly over a different element which is to be validated instead of the old one as the drop target.

How the items are highlighted as potential drop targets and how you even discover whether you're over a potential drop target depends on the nature of the item. It's still relatively simple with the items of a ListView. An item is highlighted, as we've mentioned, using the LVS_DROPHILITED attribute. When this attribute is removed from the element, it will again appear in its original form. By using the LVM_HITTEST message, it's easy to determine whether the mouse cursor is positioned over an item. The only slight inconvenience is that the specified coordinate, which refers to the upper-left corner of the parent, must first be set relative to the upper-left corner of the client area of the ListView before you can send it to LVM_HITTEST.

Converting the coordinates

You therefore have to make a slight detour by first calling the API function ClientToScreen() to convert the coordinates to absolute screen coordinates relative to the upper lefthand screen corner. For that purpose, the function receives the window handle of the dialog box as well as the coordinates. ScreenToClient() is then called, in which case you specify the handle of the ListView control along with the returned coordinates. The coordinate that is returned to you indicates the distance of the specified point from the upper-left corner of the ListView client area. This is the coordinate required for sending the LVM_HITTEST message. That's how you determine whether the mouse pointer and the drag image are now over a potential drop target.

If the state of an item below the mouse cursor and its appearance must be changed, don't forget to first make the drag image transparent. Otherwise, the original screen content will be restored again where the drag image was the next time ImageList_DragMove() is called. However that would no longer agree with the display mode of the item, as it has just been set. So, the Image List function ImageList_DragShowNoLock() is called before a state change of items below the drag image, to make the drag image transparent and to permit changing (and drawing) the item lying under it in peace. Then call the function once more to redisplay the drag image.

```

BOOL ImageList_DragShowNoLock(
    BOOL fShow                // Display drag image? TRUE or FALSE
);

```

The function expects the value TRUE as the only argument if the drag image appears and FALSE if it is to be made transparent.

The following code segment clarifies this way of responding to the receipt of a WM_MOUSEMOVE message during an active drag operation.

```

case WM_MOUSEMOVE:                // Evaluate mouse movements for drag operation
{
    if( bDragging )
    {
        int iItem;
        POINT pt;
        LV_HITTESTINFO lvhtst;

```


Besides that, the image list with the drag image generated at the start of the drag operation using `LVM_CREATEDRAGIMAGE` is released again by calling `ImageList_Destroy()`.

Finally, the capture is also released using the `ReleaseCapture()` API function, so that other windows will be able to respond to mouse events. Moreover, the `LVS_NOSCROLL` window style is again removed from the `ListView` control, enabling the visible segment to be scrolled again. With that, the drag operation is complete, and the program can continue as usual with its response to mouse and keyboard events.

Moving to the drop point

As the final result of the drag operation, the item whose number you noted at the very beginning in `iDragItem`, is moved using the `LVM_SETITEMPOSITION` message to the position above which the user dropped it. The correct coordinates, which must relate to the viewing area of the `ListView` control are not so easy to compute. It's first necessary to isolate the two 16 bit-coordinates from the Lo-word (X) and hi-word (Y) of `lParam` delivered with the message.

The coordinates are, however, relative to the upper-left corner of the dialog box (that's why it has the capture). By calling the API function `ClientToScreen()` they're first converted to absolute coordinates. For that purpose, the function receives the window handle of the dialog box and coordinates.

However, that is still not the desired format. After all, it's necessary to set the coordinates in relation to the upper-left corner of the client area of the `ListView` control. This takes place by a subsequent call to `ScreenToClient()`, in which case the handle of the `ListView` control is specified along with the coordinates just returned. What you then get back as a coordinate designates the distance of the drop point from the upper-left corner of the `ListView` client area. However, that is still not the point you must specify when sending `LVM_SETITEMPOSITION`. This function expects a view coordinate which relate to the display area of the `ListView` control. For this reason, you must still first determine the view coordinate of the point currently appearing in the upper-left corner of the client area. Use the `LVM_GETORIGIN` message or the associated macro, `ListView_GetOrigin()`. The coordinates returned here must still be added to the previously determined distance from the upper-left corner of the client area. Do you have it then? No, not quite yet! You still have to subtract the movement of the hot spot relative to the upper-left corner of the drag image, as previously set at the start of the drag operation. Then you've finally got the new view coordinates of the item, and the user will be able to move it to the desired position using `ListView_SetItemPosition()`.

```
case WM_LBUTTONDOWN:
    if( bDragging )                // Drag operation?
    {
        POINT pt, ptOrg;

        pt.x = LOWORD( lp );        // Note mouse coordinates
        pt.y = HIWORD( lp );

        // Convert dialog coordinates into ListView coordinates —————
        ClientToScreen( hWnd, &pt );
        ScreenToClient( hListView, &pt );

        ImageList_DragLeave(NULL);
        ImageList_EndDrag();         // End drag
        ImageList_Destroy( hDragImage );

        SetDropHighlight( hListView, // Drop target again normally
                          -1,         // Display
                          iDropTarget );

        // Query current upper-left corner of the ListView... —————
        ListView_GetOrigin( hListView, &ptOrg );
```

```
// ...and drop drag item at mouse position relative to origin ——
ListView_SetItemPosition( hListView,
                          iDragItem,
                          pt.x + ptOrg.x - ptHotSpot.x,
                          pt.y + ptOrg.y - ptHotSpot.y );

bDragging = FALSE;                                     // End dragging
ReleaseCapture();                                     // Release mouse

// Allow movements again inside the ListView ——
SetWindowLong() hListView,
GWL_STYLE,
GetWindowLong( hListView, GWL_STYLE) &
~LVS_NOSCROLL );
}
break;
```

Sample program

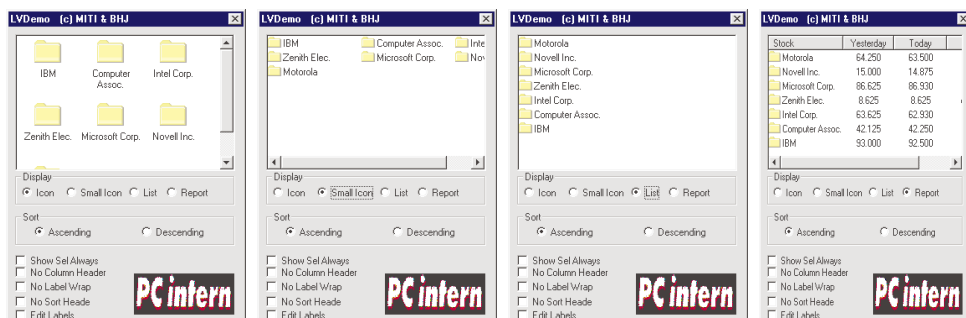
The sample program LVDEMO gives another demonstration of working with the various LVM messages for the management of a ListView control. As the following illustrations show, the program creates a dialog box with a ListView control, IDC_LISTVIEW, located at its center. The control is used to display stock market prices of various hardware and software producers. In small icon view, large icon view and list view, only the icon which is identical for all items and the name of the company appear. However if you switch to report view, the prices for the day, the previous day and the trend are also displayed in separate columns. You can also tell which direction the stock moved in comparison with the previous day the other three views, from the state image which appears next to the item's icon. It contains an arrow which identifies the trend as rising, falling or unchanged.

You'll find the following program(s) on the companion CD-ROM

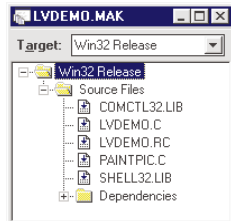


LVDEMO.C (C listing)

LVDEMO in (a) Large Icon view, in (b) Small Icon view, in (c) List view and in (d) Report view



LVDEMO in (a) Large Icon view, in (b) Small Icon view, in (c) List view and in (d) Report view



The program responds to double-clicking of items and contains a complete Drag and Drop implementation, allowing you to position the items in any desired arrangement in both icon views. Also, you can sort the items in report view according to various column criteria by merely clicking the column header. The current setting in the “sort” area of the dialog box determines whether an ascending or descending sort is carried out.



OLE (Object Linking and Embedding)

The Microsoft team of developers for the Shell is making sure that life won't get boring for programmers. Now that we've become familiar with the Win32 API, the next generation of software interfaces is making its debut: OLE objects. The interfaces for accessing the folders of the Desktop and for generating shortcuts are pointing in the direction of the future. Object-orientation per Microsoft is coming.

API functions provide only part of the system functionality. OLE objects and OLE technology offer more functionality. This is why we'll discuss OLE technology; program controlled access to the Desktop is rather limited without OLE calls. The few API functions implement only part of the functions found in the OLE objects of the Shell. For example, you cannot even generate shortcuts without resorting to the reserved OLE object of the Shell. So, in this chapter we'll talk about OLE as the basis of future object oriented system enhancements. This information will also help you understand the Windows 95 Shell and the Desktop in the next chapter.

Microsoft overwhelms its readers with more than a thousand pages of OLE documentation containing a maze of ideas. Since many of these ideas are difficult to understand, many interested users become indifferent with OLE. This is why OLE technology has gained the reputation among programmers of being too complicated and too hard to manage. OLE is indeed complex, and, above all, excessive. Contrary to popular opinion, however, OLE is not a monolithic block. Instead, it's a blend of objects and ideas. You don't even need to be familiar with all the objects and ideas.

Many secrets of OLE are revealed when the basic meaning of the whole and its achievement at code level in C and C++ become clear. You only have to examine a fraction of the concrete OLE specification to be able to harness this technology as a client. So, let's get our feet wet with OLE.

OLE Compact Seven Steps to accessing OLE objects

- OLE objects are located in EXE or DLL files. You can recognize OLE objects by a unique class ID (CLSID). The ID serves like an "official" label.
- OLE objects include servers and clients. Servers are OLE objects that offer a specific function and clients use these objects. Clients can be applications or other objects.
- The functions of an object are revealed by its interface. The object has no additional functions other than the interface.
- Each interface contains a logically related group of functions (interface methods) and has a unique interface ID (IID).
- Besides its methods, each interface has the IUnknown interface with three methods. The client can "bind with" other interfaces using the IUnknown interface and call their methods.
- An object exists only through its object instances. An API function is provided for most objects. It can be used to generate a new object instance and be returned as a pointer to the interface contained within the new object instance.
- Besides its data, each object instance contains a pointer to the vtable, the (virtual) function pointer table with the addresses of the individual interface methods. You can call these methods like normal functions in C and C++ using this table.

OLE's Roots And Development

Microsoft's OLE is a perfect example of how computer technology can rapidly change. Today, when you hear the term OLE, you probably think an Excel or Word for Windows document into which you are linking a graphic, a form or something similar (in other words, an object). After all, doesn't OLE mean "Object Linking and Embedding?" This definition is not necessarily wrong but it hides what OLE is all about and what needs to be emphasized. This is the basic idea of how objects and their functionality are implemented and how access to them is handled. We're talking about the *Component Object Model* (COM). The ability to link graphics and other elements to a Word for Windows document is only a demonstration of a very general principle that is important here.

The "Component Object Model" describes a binary standard with whose help code modules to objects can be organized, which pass their functionality on standardized paths to the outside. Objects stored in a file as objects in COM format can be directly called from C or C++ programs. What people commonly understand as OLE is merely a small piece of a larger puzzle, not to be confused with the puzzle itself. Other puzzle pieces will achieve greater importance in the future than classical "linking and embedding." For example, automation objects are required for accessing OLE objects from Visual Basic, Delphi or Microsoft Excel and Access. However, automation objects are also only pieces of the puzzle, which is inadequately characterized by the name of OLE.

Successor to Dynamic Link Libraries (DLL)

We could consider COM objects as the DLLs of the 21st century provided Microsoft doesn't develop something else again. COM objects are the next step up the evolutionary ladder from Interrupt call to superdynamic object. (A superdynamic object will "fly" around through cyberspace and be available to anyone with the correct password or the proper permission.) Although we're not that far yet technically, the trend is definitely leading in that direction. Static or early binding to runtime libraries at compile time followed the interrupts. Dynamic or late binding to DLLs at runtime arrived next. Now, it's OLE objects according to the COM standard. With objects increasing in popularity thanks to C++ and class libraries per MFC (Microsoft Foundation Class), we certainly cannot ignore the operating system. We need to view objects as the basic building block on the path to object oriented operating systems and applications.

Programmers haven't forgotten DLLs yet. However, in areas where the system is tapping into new functionality, this functionality is becoming more accessible to programs through OLE objects. You can access these objects directly in C and C++. Visual Basic and other VBA environments require OCX objects for this purpose. These objects fit over the existing OLE objects. This gives them the appearance required for linking them in VBA environments.

The basic goal of the COM specification is universal applicability and interchangeability of objects. A PC program will access an object offered to it without noticing that this object is not being executed on the same machine, but rather on a different computer on the network. That's still not enough, however. The executing machine doesn't even have to be an Intel PC. It could also be a MIPS computer under Windows NT or (sometime in the future) an Apple, provided the Apple operating system supports the COM standard. This is precisely what we mean by a Client/Server system.

The client is the caller or user of the COM object. The server is the COM object itself. Whether this type of client-server application makes sense for normal applications in the office today and what the performance of such an object call will be like on a network, are entirely different questions. However, what is certain is that the COM interface builds the foundation for transferring functionality from one's computer to any server on the network. It doesn't matter whether the server is in the next room or on the other side of the world. Currently, you can only work on objects that are on the same computer and being executed at the same time, like the client. However, the trail toward future developments has already been blazed.

At system level, you'll see OLE as an OLE subsystem in the OLE32.DLL file. This file contains about 150 API functions that help you in working with and accessing OLE objects. Some API calls are required before starting your object oriented business. Then you get an object with which you can work.

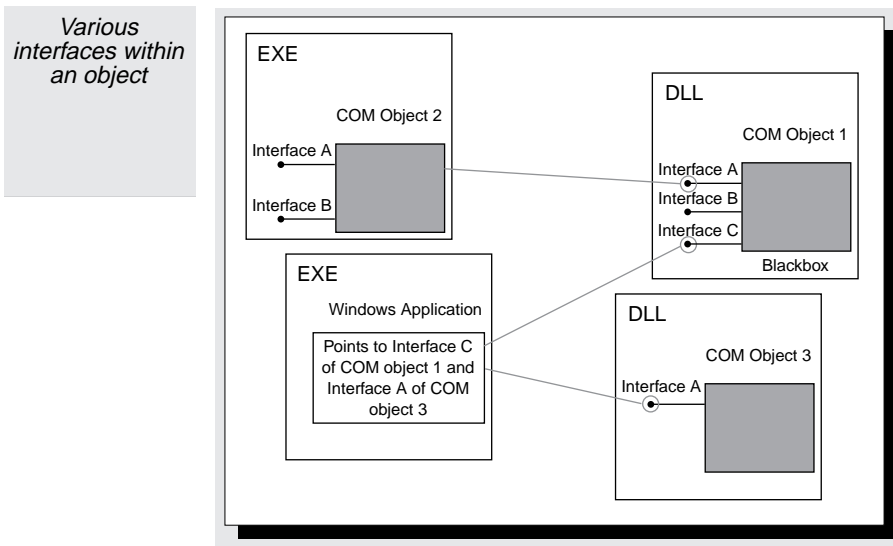
Several Include files are available for calling OLE functions at C level. These Include files must be linked in the OLE applications. First, OLE2.H links OBJERROR.H, OBJBASE.H and OLEAUTO.H. OBJBASE.H contains the underlying declarations. Don't be shocked by the large number of definitions and the extensive use of preprocessor commands. You won't

need most of the definitions or the preprocessor commands. Using many of the remaining definitions and preprocessor commands are easier in concrete programming than you might think. Also, the OLE32.LIB file is available for calling the OLE function in a C program.

Interfaces and their implementation

You need a clear definition of the interface that is visible to the caller for a transparent usage of objects. This is true regardless of whether objects reside on a server on the network or on the local machine. An object must show which possibilities it offers a client and how the client can access them.

On the other hand, objects don't need to concern themselves with the identity of the user (Client) and what is occurring beyond the interface boundary on the client's side. It's a black box if observed from the outside. You don't see what occurs inside but only what goes in and out of the interfaces. Also, the outside world seems to be a black hole to the object. Remember, its small "universe" ends at its boundaries. The narrow boundary between inside and outside is formed by the interface in COM format. The outside world communicates with the object through the COM format interface.



Interfaces, according to the COM definition, pass a COM (you could also say an OLE) object to the outside. If an API function is the smallest unit that exports a DLL, with OLE objects, the interface assume this role. An interface is a collection of logically connected functions that can be used by an external caller.

The COM specification determines how interfaces are defined, which opportunities they offer and how they appear on binary level. This establishes a framework from which interfaces and functionality can be defined using an abstract description. However, until you fill this framework, and until you implement the methods of an interface through concrete program code, it's all basically meaningless, just a theoretical construct.

This where one of the many misunderstandings about OLE starts. Inevitably, the following questions are asked: "Who defines concrete OLE interfaces? Who implements them? Who uses them as a client?" The answer to these questions is a rather confusing "everybody." Everybody can define OLE interfaces according to the COM specification. The system, each Windows application, Visual Basic, Corel Draw, etc. can all define OLE interfaces. The only questions are: "Who converts these interfaces to concrete program code?" and "Who uses them?" In other words, who plays the server and who plays the client in the end.

In practice, you'll meet with two sets of circumstances:

1. Someone defines a specific interface and implements as an OLE object so other applications can access it. The OLE system, for example, defines some OLE interfaces such as for saving OLE objects in files. The interfaces are documented so a caller can use them according to the COM model. The “caller” (Client) can be an application, the system or independent OLE objects. It’s like the functions for accessing the Desktop. They’re also defined as OLE objects by the Shell and, therefore, exposed to the call.
2. Someone defines an interface, but leaves the implementation to other OLE objects. This is usually interfaces required for communication between objects. For example, it could be the interfaces that an OLE object must bring along so it can be embedded in a Word document or be recognized by a VBA environment. The object uses this interface to communicate with its container, in this case Word for Windows or Visual Basic. Because the interface for linking the objects is clearly defined, anyone can implement such an object. Therefore, many implementations of the interface are possible within different objects. When the user designates an object for linking, the server makes certain that it contains the required interfaces and calls them. All interfaces look alike from the outside. However, they contain unique code on the inside. This code depends on their purposes. This is one of the basics of object oriented design called “polymorphism.”

With Shell programming under Windows 95, the first scenario is emphasized. You use existing interfaces but without having to develop custom OLE objects or implement predefined interfaces. You act only as the client, not as the server. Ninety percent of the problems with OLE simply disappear. Topics such as creating an object, the derivation, inheritance, aggregation or definition of the interface are unimportant. You’ll only come across these topics if you want to act as the server, but that’s a different story.

The following table summarizes the more than 50 interfaces that the OLE subsystem defines. Fortunately, you’ll probably need only few, for example, only two are necessary for desktop programming.

Predefined OLE interfaces	
Interface	Task
IUnknown	The Standard Interface
IClassFactory	Creates new object instances
IOLEClientSite	Informs embedded objects about their container.
IMalloc	Allocates dynamic memory
ILockBytes	Manages a byte array for receiving an object.
IPersistFile	Saves or loads an object from a file.
IDispatch	For Automation Controls (e.g.; OCX).
IStorage	Controls access to “Structured Storage” or “Compound Files”

In the chapter on the Shell and the Desktop, OLE is used exclusively as a further development of the DLL idea, for calling a ready-made functionality. However, the DLL functions don’t disappear completely. For example, the Shell exports some API functions, which, along with the OLE interfaces, will continue to be required. For example, you need API functions to get a Shell object that will allow you to use the functionality of the Shell OLE interface.

Because you deal with OLE interfaces slightly differently than API calls and DLLs, the following sections summarize the most important concepts and conventions necessary for calling (Shell) objects from C.

Dynamic binding

The OLE interface versus the DLL function is not the only difference between the OLE concept and DLLs. It’s also the way you bind with this object and its interfaces; in other words, how the application connects to the object and calls its services. The OLE concept resembles dynamic binding with a DLL at runtime using LoadModule() and GetProcAddress(). The connection between client and server with OLE is also made first at runtime. This is also why the linker doesn’t require any information about the location of the OLE objects that you want to use.

This is all arranged at runtime. System components that furnish OLE objects normally also export some few API functions. Among these few API functions, there is always one that creates an object of the desired type. As the caller, you get a pointer to the new object instance. It will be used for all future accesses to the object.

Objects are represented by their interfaces. Interfaces group logically related functions into a solid unit. For example, all operations that process a specific type of object, delete it, search for it or whatever else can be done to it. Therefore, interfaces usually consist of fewer than twenty functions instead of hundreds of functions.

While a series of interfaces has already been defined and implemented by the OLE system and other system components, you can add new ones at any time. Whoever wants to develop a new OLE object writes and implements their interfaces. Depending on the desired functionality, an object will contain several interfaces predefined and self-defined. The precise number of predefined and self-defined interfaces must be combined to provide the desired functionality. Therefore, the type and number of interfaces vary depending on the object. However, keep this in mind: When an interface is made available, all the functions defined within must be implemented. The client must always be able to rely on this rule.

Due to the wide variety of interfaces, a dynamic mechanism was created for binding with an interface, which makes use of the interface concept itself. Regardless of its task, every COM object must provide at least one predefined interface called the IUnknown interface. It serves as a starting point for potential clients and helps clients determine whether the object in question provides a desired interface. The equivalent function is called QueryInterface(). When you get a pointer to an object, it means you're holding a pointer to an IUnknown interface in your hands. To determine whether a desired interface is implemented, call QueryInterface() using the pointer.

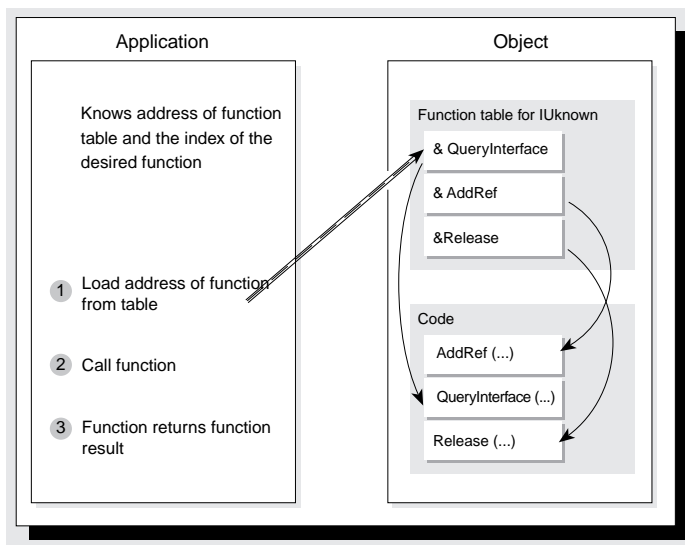
You have a pointer to the interface if the call succeeds. Use the pointer to call the functions of the interface at any time just like normal API functions. If the call to QueryInterface() fails, the object does not provide the desired interface.

Calling interface functions

In this chapter we'll also see how interfaces are defined at C level and how, as a result, the call for interface functions is specified. However, right now we'll take a short look at the realization of interfaces, to make the term interface slightly less abstract.

At the lowest level, an interface is simply a table of function pointers. Since each entry in the table represents one of the interface functions, you could also say that each entry represents one of the methods of the object. So, a caller only needs to know a pointer to this table and the index of the desired functions to be able to call them directly and without obstacles, like any other C function.

Calling interface functions using IUnknown as an example



However, what is returned from `QueryInterface()` or one of the many functions from the OLE API is not the function table itself but a pointer to an object instance. You might say it's a reference to the living object. It contains the data that every class instance possesses. In Visual Basic you would refer to the data as the properties of the object, in C++ it would be the member variables of the object. However, you don't know where they are or how they are structured. Unfortunately, that's how it must be with object oriented programming (buzz word: "encapsulation"—the encapsulation of data and program code).

Here's what you do know: The returned object pointer refers to the beginning of the (object) structure in memory, precisely where a pointer to the function table representing the interface of the object is located. There may be other interface pointers in front or in back of it, because objects can contain several interfaces, but you don't know this. You can easily address the returned interface by seeing the returned pointer as a pointer to a pointer which refers to the function table with the addresses of the individual interface methods. Through double dereferencing/callback/double indirect addressing you can get to any interface function immediately and call it like a normal C function. It appears like the following in C:

```
x = InterfacePtr->FunctionTable->MethodInTheFctTable( parameter );
```

Naturally, you could ask why the table with the function pointers (the actual interface) isn't stored directly with the object. That would have saved one indirection/dereference/callback in the call. The reasons why you should not do this are twofold:

1. This would waste a great deal of disk space. Unlike their data, all objects of a specific object type share a common set of interface functions. Why store an identical table with function pointers in one thousand different object instances when a single copy is enough?
2. This controls the overwriting of virtual functions in the derivation of object classes, for example in C++. Simply change the pointer of one of the interface methods in the function pointer table and all the clients of the interface will be calling the new method instead of the old method. This occurs completely transparent; the clients don't contribute anything.

OLE and C++

That brings us to our subject. Object classes in C++ and OLE objects are fundamentally different. Some concepts are clearly deviate from each other (for example, in the question of inheritance). However, on one level the concepts are quite similar. COM objects, like C++ objects, are stored in memory. Alone the name of the pointer to the actual interface with the jump pointers/go to pointers makes this clear. This pointer is called `lpVtbl`, borrowed from the `Vtable` of a C++ object, which refers to the methods of the object, just like the function pointer table of a COM object. In reference to OLE, you would say: to the functions of the interface. There are different names for things, but at the lowest level, the concepts are identical.

C++ offers a decisive advantage for calling OLE interfaces: It already contains the syntactical constructs which it requires for accessing an OLE object. For this purpose, interfaces like `IUnknown` are derived from a C++ class by means of corresponding Include files:

```
class IUnknown
{
public:
    virtual HRESULT QueryInterface( const IID &riid, void **ppvObject );
    virtual ULONG AddRef( void );
    virtual ULONG Release( void );
};
```

The call takes place then through the usual C++ syntax:

```
class IUnknown ciu; // Object of Iunknown type
HRESULT hr;
ULONG ul;

hr = ciu.QueryInterface( parameter ); // Call QueryInterface method
ul = ciu.Release(); // Call Release method
```

Furthermore, when calling in C++ save yourself the trouble of the implicit this pointer which, when calling from C, you always have to specify as the first parameter when calling an interface function. An interface (or member) function requires this pointer to be able to access the data within the object instance. Therefore, such a pointer is automatically passed in C++ as the first parameter with every call for a method. It doesn't need to be defined explicitly. Within the program code of a method, however, you can expressly address the pointer to refer to the object. This is the same pointer through which you read out the address of the function pointer table in C, that is, the address of the lpVtbl variable. Because the variable is at the beginning of the object in memory, it embodies the object itself. In comparison to calling an interface function in C++, when calling in C you have to specify one parameter more, however, it's always the same one: The one through which you access the lpVtbl at the beginning of the object.

The following applies for C++:

```
MyObject.QueryInterface(riid,ppvObject)
```

The following applies for C:

```
MyObject->lpVtbl->QueryInterface(MyObject,&riid,&ppvObject)
```

Even if it is only a matter of more typing - the advantages of using C++ for working with OLE objects are indisputable. Nevertheless, we're betting on regular C for several reasons in this chapter: It does a better job of making the basic mechanism of the OLE call clear. Anyone who understands it on C level will have no problems using it under C++. However, anyone who doesn't know C++ will hardly be able to derive the call in C from it. What is more, when using C++ in conjunction with OLE, you are more or less gently driven into the arms of a class library like the MFC. That makes the whole business not only needlessly complicated, but also artificially inflates it. Our maximum value for creating an OLE object with no functionality including of the MFC amounts to 180K. It's no joke! That made it easy for us to choose C.

Identification of an interface

Let's turn our attention back to IUnknown. When working with OLE, you'll frequently see identifiers like this one because all interfaces have names starting with I. This includes the more than 50 interfaces predefined by OLE. These have names like IMoniker, IOleInPlaceFrame or IDispatch.

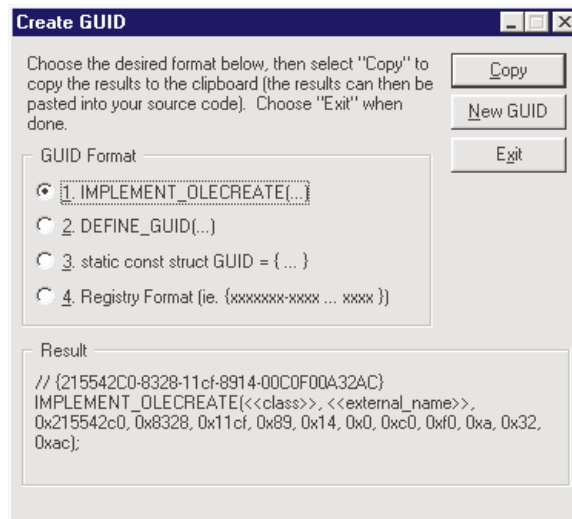
The names are important. To be able to query an interface using QueryInterface(), the interface must have a name or other type of identifier that clearly separates it from all other interfaces. As long as you are only making do with objects of the current machine, these interface identifiers only need to be unique within the machine. However, if you are already thinking in the categories of extraterrestrial networks, as with the COM concept, then the interface identifiers have to be unique on a global, almost universal basis. That's exactly what these identifiers are called: GUID (Globally Unique Identifier).

It's not the symbolic name, such as IUnknown, but rather its GUID that represents an object on the lowest level. GUIDs are 128 bit (4 DWORD) data structures that uniquely represent an interface. Anyone who creates a new interface generates their GUID using a tool from the Win32 SDK called GUIDGEN.EXE. This GUID is globally unique and will represent the interface from that point. This is important because with each call by any user at any given time, the program, according to the character of a GUID, must always create a unique GUID, i.e., ultimately another 128 bit number, as it did with the subsequent million calls. This also occurs without a central instance that could coordinate the execution of the countless program calls.

Although Microsoft hasn't disclosed exactly how this program operates when generating a GUID, we do know how to achieve such a thing. To generate a GUID on a machine, simply collect enough values with the most random and most quickly changing origin possible. Examples of this could include the contents of random memory locations in RAM, the tick of the system clock, the number of free bytes on the hard drive, the name of the user, and since the Microsoft stocks achieved a new all-time high. Simply all the random events that you can find out on the current machine at program runtime. It's very helpful in such cases if the machine has a component with a unique ID number. For example, this applies to network cards whose manufacturer has been using centrally managed allocations of ID numbers for years and burning these numbers into their cards. Each network card is thus guaranteed to differ from its neighbors and makes certain your GUIDs are unique.

The gathered random values are linked together by a mathematical formula or an iterative algorithm so that the formula results are spread out. If only one bit changes in one of the many random parameters from call to call, you get entirely different GUIDs.

*The
GUIDGEN.EXE
program*



You're hoping eventually for a *chaos function*. It consists of sequences of function results that will never be reproduced because the input parameters will never occur in the same constellation.

For example, the {93142420-F35A-11ce-AD52-0000E8C841A6} GUID was created in this illustration. The hex display in braces is defined within the COM specification and reflects the internal structure of a GUID in the sequence DWORD, WORD, WORD 8 bytes.

The GUID data type is defined in the OBJBASE.H file as a sequence of one DWORD, two WORDs and eight bytes. The meaning contained in the

individual fields is unknown. Various other data types are derived from GUID. Although they also represent “globally unique identifiers”, they’re used for special purposes. CLSID is used as a class ID for unique naming of OLE objects and IID is used for unique naming of interfaces. RIID also represents a GUID.

LPIID and LPCLSID define pointers to CLSIDs or IIDs in memory. Finally, IID_NULL and CLSID_NULL indicate invalid class or interface IDs.

```
// commented excerpt from OBJBASE.H

typedef struct _GUID          // the wonderful, unique Mr. GUID
{
    DWORD Data1;             // 4 bytes
    WORD  Data2;             // 2 bytes
    WORD  Data3;             // 2 bytes
    BYTE  Data4[ 8 ];        // 8 bytes
} GUID;                      // 16 bytes in total = 128 bits

typedef GUID IID;            // IID is an interface ID
typedef GUID CLSID;          // CLSID is a class ID
typedef GUID RIID;           // RIID is a general "Reference ID"

typedef IID __RPC_FAR *LPIID; // FAR pointer to IID
#define IID_NULL GUID_NULL    // invalid interface ID
typedef CLSID __RPC_FAR *LPCLSID; // FAR pointer to CLSID
#define CLSID_NULL GUID_NULL   // invalid class ID
```

Using GUIDs

GUIDs are used to identify objects as well as interfaces. As every interface has an interface ID, each object also has its own class ID. Class and interface IDs are managed separately because objects can contain several interfaces, possibly implementing the same interface. Therefore, class and interface IDs are required in different locations:

For one, they are required within programs that want to access an interface in an instance of a specific object as clients. So, the developer of an object packs information about the object and its interfaces in the Include file, which you require for accessing an interface in C or C++ at lowest level as a client. Just as you need Include files with prototypes, data structures,

types and constants to use DLL functions, the author of an interface also passes on class and interface IDs in the form of Include files to developers. Also, the structure of the implemented interfaces is defined in the Include file. A potential client, when calling the QueryInterface() function in the IUnknown interface, can use the specified interface IDs to get the desired interface.

Also, the class and interface IDs are required within the program code which implements the object with its interfaces, in the source code of the server, as it were. Within the QueryInterface() function to be made available, you check the interface ID passed from the client and compare it with your own. If it doesn't match up, the client has "the wrong number" in the truest sense of the word, and no pointer to an interface will be returned.

Finally, at least the class ID is required when an OLE control makes its mark in the registry. OLE controls are stored in EXE or DLL files, which is why the system has to build a connection between a class ID and the file which contains the program code of the object. That's what the corresponding sections in the registry (HKEY_CLASSES_ROOT) are there for.

Structure of IUnknown

When QueryInterface() is called, an interface ID is required to identify the desired interface. However, QueryInterface() is not the only function provided by the IUnknown interface. Two other functions are implemented, with the names of AddRef() and Release(). They're used for object management.

Methods of IUnknown	
Method	Task
QueryInterface	Queries interfaces and returns pointers to interfaces.
AddRef	Informs the object about a new user
Release	Informs the object of the release of a reference and destroys the object if no other users access the object

Object management is a matter of giving the object the opportunity to register the number of its users. This is important if we don't want the object to stay in memory forever and continue allocating resources. If the users log off from the object, it knows when the last user is gone, and it no longer has to remain in memory. Therefore, objects manage reference counts, which are controlled externally using AddRef() and Release(). Each new user must log on using AddRef() and log off using Release() before the object is terminated.

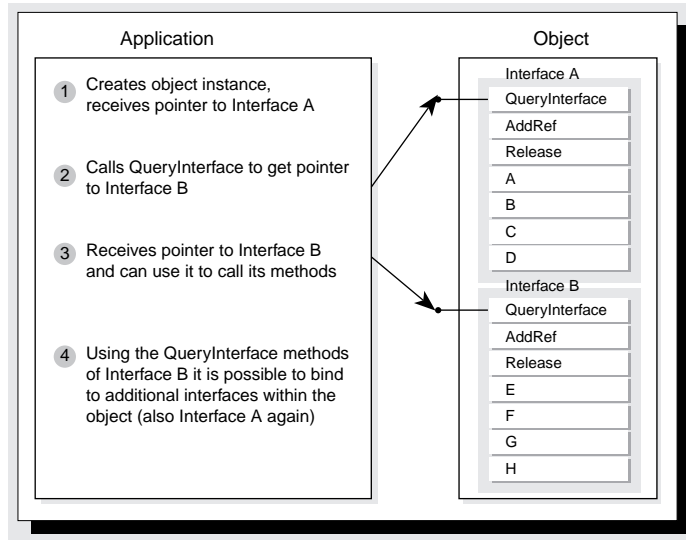
If you use a default API function to generate a new object instance of an object (for example the OLE API function CoGetMalloc()), this function returns a pointer to a predefined interface of the object and has already called AddRef() from this interface. So the caller only needs to execute the Release() call before he terminates the object or no longer requires it. We'll cover this in detail again when we describe the various Shell interfaces.

It's important to understand that each interface of an object must have its own IUnknown interface. If an object contains ten interfaces, from the callers point of view, along with each interface, there are ten IUnknown interfaces available. You can gain access to all the other interfaces of an object through each of these IUnknown interfaces. So it's not necessary to note the original pointer to get to a QueryInterface() function for accessing a different interface. Every pointer that refers to an interface can also be used to call QueryInterface(). Thus you can gain access to all other interfaces of an object through any interface pointer.

Let's take a closer look at the three Iunknown methods:

```
HRESULT QueryInterface(
    void *this                // pointer to the object
    IID *iid,                 // interface ID of desired interface
    void **ppvObject          // pointer to Var that gets the interface pointer
);
```

*Using
QueryInterface(),
AddRef() and
Release()*



After the mandatory this pointer, QueryInterface() expects the address of a variable with the interface ID of the interface you wish to access. In other words, this is the interface that the QueryInterface() call is to provide. In the third parameter, the address of a variable is expected in which QueryInterface() will place a pointer to the desired interface in case of success. In case of error, you will find NULL there. However, you will be able to tell whether the interface is available from the function result, which, as with many interface methods, is of the HRESULT type. Sometimes SCODE is specified instead of HRESULT, however both types are identical under Win32, therefore they are interchangeable. Both represent a 32 bit status value with the following structure

Unlike using Boolean or integer values, through the classical division of error or success, an SCODE makes possible a finer differentiation in the form of various groups of error codes and the error or success code contained within. Those who aren't concerned with the details only need to look at bit 31 of an SCODE, while those who are interested in more extensive error information will find it in the other two fields.

Structure of an HRESULT and SCODE.	
31	Indicates error (1) or success (0).
16-30	Error group
0-15	Actual error or success code in the context/frame of its error group

Several constants have been defined for possible error and success codes, however, you only need to be familiar with a few. If you're only concerned with success and failure, the two macros defined in OLEBASE.H, SUCCEEDED() or FAILED(), which want to be fed with an HRESULT or SCODE, are helpful. They return TRUE when the HRESULT or SCODE indicates success (with SUCCEEDED()) or failure (with FAILED()). Otherwise FALSE is returned.

The constants indicating a successful function execution usually begin with S_ (for success), while those that indicate errors start with E_ (for error). Sometimes these constants will also have prefixes that indicate the error class, for example OLE_E_NOTRUNNING or MK_E_LAST. The same holds true for success codes like CLIPBRD_S_FIRST or CO_S_LAST. You will find hundreds of these codes in the winerror.h include file. You need to remember a few of them:

The most important HRESULT codes	
S_OK	Shows affirmation of the question with functions such as IsValid() or IsString()
S_FALSE	Shows negation of the question with functions such as IsValid() or IsString()
E_NOINTERFACE	QueryInterface() cannot return the desired interface
E_FAIL	General error
E_NOMEMORY	Not enough memory to execute the desired operation
E_NOTIMP	The called interface method was not implemented

The following code excerpt shows an attempt at getting an IShellLink interface through a given pointer, accessing its Release method, which it contains using the required IUnknown interface.

```
LPUNKNOWN ppunk;      // pointer to Iunknown interface
LPSHELLINK pshl;      // pointer to IShellLink

// ppunk points to the Iunknown interface of an object,
// whose IShellLink interface is to be determined and
// called

if ( SUCCEEDED( ppunk->lpVtbl->QueryInterface(ppunk, &IID_ShellLink, &pshl)))
{
    HRESULT hr;

    printf( "pointer to IShellLink received\n" );

    hr = pshl->lpVtbl->SetPath( parameter );           // call interface method
    hr = pshl->lpVtbl->SetHotKey( parameter );         // call interface method

    // more calls for IShellLink methods

    pshl->lpVtbl->Release( pshl );                     // release interface again
}
```

AddRef() and Release()

The two complementary methods AddRef() and Release() are defined as follows:

```
ULONG AddRef(
    void *this           // pointer to the object
);
ULONG Release(
    void *this           // pointer to the object
);
```

Although both methods expect only this as a parameter, they return the previous value of the reference count. Don't read too much into this value because it doesn't have any significance for normal work with the object. It's mainly used for debugging and test information.

Declaration of interfaces in practice

We're going to introduce you to the various interfaces of the Shell later in the Desktop section/chapter. With such a definition in plain text, it's not difficult to call OLE interfaces (as easy as operating normal API functions). However, sometimes only an Include file is available, which makes an interface declaration and contains the constants, types and functions for accessing

the interface. You will have some difficulty at first in interpreting the interface declaration, at least in cases where this file comes from Microsoft. The declaration usually consists of macros. You must be familiar with these macros to make any sense out of the whole. This involves the concept of having a single Include file with interface declarations for binding to C and C++ programs. Depending on the language, an interface declaration that clearly differs in syntax must come out of this, which is why the macros in the interface definition are mapped to different identifiers, depending on the language.

We'll use the IShellLink interface as declared in the SHLOBJ.H Include file as an example. IShellLink will play an important role later in the Desktop chapter. It provides the functionality for generating and operating with shortcuts.

```
// Declaration of IShellLink interfaces in SHLOBJ.H

#undef INTERFACE
#define INTERFACE IShellLink

DECLARE_INTERFACE_(IShellLink, IUnknown)
{
    // *** IUnknown methods ***
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID * ppvObj) PURE;
    STDMETHOD_(ULONG,AddRef) (THIS) PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;

    STDMETHOD(GetPath)(THIS_ LPSTR pszFile, int cchMaxPath, \
        WIN32_FIND_DATA *pfd, DWORD fFlags) PURE;

    STDMETHOD(GetIDList)(THIS_ LPITEMIDLIST * ppidl) PURE;
    STDMETHOD(SetIDList)(THIS_ LPCITEMIDLIST pidl) PURE;

    // a series of other interface methods follows
    // in the same style

    STDMETHOD(Resolve)(THIS_ HWND hwnd, DWORD fFlags) PURE;
    STDMETHOD(SetPath)(THIS_ LPCSTR pszFile) PURE;
};
```

DECLARE_INTERFACE, STDMETHOD, THIS_ and PURE are preprocessor macros that are converted during compiling. The following are their definitions for C:

```
#define interface struct
#define STDMETHOD(method) HRESULT (STDMETHODCALLTYPE * method)
#define STDMETHOD_(type,method) type (STDMETHODCALLTYPE * method)
#define PURE
#define THIS_ INTERFACE FAR* This,
#define THIS INTERFACE FAR* This
#define CONST_VTBL const
#define DECLARE_INTERFACE(iface) typedef interface iface { \
    const struct iface##Vtbl FAR* lpVtbl;\
} iface; \
typedef const struct iface##Vtbl iface##Vtbl; \
const struct iface##Vtbl
```

The following are their definitions for C++:

```
#define interface struct
#define STDMETHOD(method) virtual HRESULT STDMETHODCALLTYPE method
```



```

#define STDMETHODCALLTYPE virtual type STDMETHODCALLTYPE method
#define PURE = 0
#define THIS_
#define THIS void
#define DECLARE_INTERFACE(iface) interface iface
#define DECLARE_INTERFACE_(iface, baseiface) \
    interface iface : public baseiface

```

Regarding IShellLink, the following declarations result for C: IShellLinkVtbl as pointer to the Vtable of the IShellLink interface with the function pointers to the individual interface methods, as well as IShellLink, which represents an object of the ShellLink type with the pointer lpVtbl of the type IShellLinkVtbl. Keep in mind that all the methods actually expect as the first parameter a this pointer which must point to the object in question and thus, in this case is always of the IShellLink* type. As a rule, a corresponding pointer type is defined for the objects predefined by OLE. It gets the name of the interface without the I at the beginning, but does have the LP prefix, which represents long pointer. In this case, the pointer type is named LPSHELLINK.

```

typedef struct IShellLink // Object of IShellLink type
{
    struct IShellLinkVtbl * lpVtbl; // Pointer to VTable
} IShellLink;

typedef struct IShellLinkVtbl IShellLinkVtbl;

struct IShellLinkVtbl // the VTable of IShellLink
{
    HRESULT (* QueryInterface) (IShellLink * This, const IID * const riid, \
                                LPVOID * ppvObj) ;
    ULONG (* AddRef) (IShellLink * This);
    ULONG (* Release) (IShellLink * This);
    HRESULT (* GetPath) (IShellLink * This, LPSTR pszFile, int cchMaxPath, \
                        WIN32_FIND_DATA * pfd, DWORD fFlags) ;
    HRESULT (* GetIDList) (IShellLink * This, LPITEMIDLIST * ppidl) ;
    HRESULT (* SetIDList) (IShellLink * This, LPCITEMIDLIST pidl) ;

    // a series of other interface functions follows
    // in the same style

    HRESULT (* Resolve) (IShellLink * This, HWND hwnd, DWORD fFlags) ;
    HRESULT (* SetPath) (IShellLink * This, LPCSTR pszFile) ;
};

```

If you have a function named CoGetIShellLink() that returns a pointer to an IShellLink object, call its methods in the following manner:

```

IShellLink *MyShellLink;

MyShellLink = CoGetIShellLink(); // get pointer to object
if ( MyShellLink != NULL ) // it's a pointer, okay?
{
    // call interface method resolve
    if (SUCCEEDED( MyShellLink->lpVtbl->Resolve( MyShellLink, hwnd, flags)))
        printf( "Call for IShellLink::Resolve successful\n");
    // it's easier with a macro
    #define IShellLinkResolve( me, h, f ) (me)->lpVtbl->Resolve(me, h, f)

```

```

    if (SUCCEEDED( IShellLinkResolve(MyShellLink, hwnd, flags)))
        printf( "Call for IShellLink::Resolve successful\n");
}

```

In the `printf()` command we use the form for addressing interface methods in C++ outside of their scope, with the notation: `Object::Method`. Although we give preference to the interface call in C, in the text we make use of this notation over and over again. It's simply the most concise form for describing an interface method precisely. Since we're already on the subject, here's what the `IShellLink` declarations for C++ look like:

```

class IShellLink : public IUnknown
{
public:
    virtual HRESULT QueryInterface ( const IID & riid, LPVOID * ppvObj) = 0;
    virtual ULONG AddRef (void) = 0;
    virtual ULONG Release (void) = 0;

    virtual HRESULT GetPath(LPSTR pszFile, int cchMaxPath, \
                           WIN32_FIND_DATA *pfd, DWORD fFlags) = 0;

    virtual HRESULT GetIDList( LPITEMIDLIST * ppidl) = 0;
    virtual HRESULT SetIDList( LPCITEMIDLIST pidl) = 0;

    // a series of other interface functions follow
    // in the same style

    virtual HRESULT Resolve( HWND hwnd, DWORD fFlags) = 0;
    virtual HRESULT SetPath( LPCSTR pszFile) = 0;
};

```

The member functions are declared here, however, without being implemented. Because the compiler doesn't allow it, the Microsoft specific addition `= 0` is set after each declaration. This ensures the compiler is satisfied with only the declaration, and you can still use classes of the type. To be specific, it ensures that the compiler also constructs the Vtable as desired, without whose knowledge the compiler won't be able to channel any calls of interface methods through an interface pointer. By the way, it's unnecessary to explicitly specify the this pointer because it's supplied by the C++ compiler automatically.

Here's what accessing an `IShellLink` object in C++ looks like:

```

class IShellLink *MyShellLink;

MyShellLink = CoGetIShellLink();           // get pointer to object
if ( MyShellLink != NULL )                 // it's a pointer, okay?
{
    // call Resolve interface method
    if (SUCCEEDED( MyShellLink->Resolve(hwnd, flags)))
        printf( "Call for IShellLink:Resolve successful\n");
}

```

Important OLE functions and classes

In this chapter we've already introduced a few functions from the OLE API. You'll need a few more if you want to use OLE interfaces in your applications.

Important functions from the OLE API for accessing OLE objects	
CoInitialize	Initializes the OLE subsystem in relation to your application
CoUninitialize	Switches the OLE subsystem off again
CoCreateInstance	Creates a new instance of an OLE object
MultiByteToWideChar	Generates UNICODE string from ANSI string
WideCharToMultiByte	Generates ANSI string from UNICODE string

CoInitialize() and CoUninitialize() are easy to operate. Call CoInitialize() before your first access to an OLE function. Call CoUninitialize() after your last access but no later than the end of the program. The function result is of the HRESULT type with CoInitialize(), so that you can use the FAILED and SUCCEEDED macros to determine whether the OLE subsystem was successfully initialized as desired.

```

HRESULT CoInitialize(                // Initializes OLE subsystem
    LPVOID pvReserved                // reserved: NULL
);
void CoUninitialize(void);           // switches OLE off

```

The call for CoCreateInstance(), which also returns an HRESULT, is a bit more trouble. While API functions such as CoGetMalloc() generate specific object instances, you can create any desired object instances using CoCreateInstance(). All you need is the class ID of the object and the interface ID of the desired interface. To do this, pass the address of an interface variable, which is to load the function with a pointer to the new object instance, in the ppv parameter. If the function call succeeds, the specified variable will point to the desired interface afterwards. Because this interface, like all others, contains the methods of the IUnknown, you can bind with to all other interfaces of the object from it, provided any exist.

```

HRESULT CoCreateInstance(             // creates an object instance with interface
    REFCLSID rclsid,                 // class ID
    LPUNKNOWN pUnkOuter,             // NULL
    DWORD dwClsContext,              // specifies the place of execution for the
                                     // object
    REFIID riid,                     // interface ID
    LPVOID FAR* ppv                  // pointer to Var that is to receive the interface
                                     // pointer
);

```

The dwClsContext parameter determines whether the OLE object is executed within the framework of an external process, or exactly like a DLL, is inserted in the memory of the client. Because the latter is significantly faster, specify the CLSCTX_INPROC_SERVER constant to set it.

The only question now is where to get the class and interface IDs that you need to successfully call the function. You'll normally find external declarations to variables in the Include files for interfaces like:

```

extern const GUID CLSID_ShellDesktop; // for the ShellDesktop object
extern const GUID CLSID_ShellLink;    // for the ShellLink object
extern const IID IID_IPersistFile;    // for the IPersistFile interface
extern const GUID IID_IShellFolder;   // for the IShellFolder interface

```

By using these external declarations, the client can gain access to variables that the OLE server provides in its DLL or EXE file and initializes with the class or interface ID. You pass the address of the variables to CoCreateInstance(). A variable is available for each class whose name begins with CLSID_. For example, the figure above shows us the class IDs for the two objects (ShellDesktop and ShellLink) the Shell exports for accessing the Desktop. The interface IDs have IID_ as a prefix as you can see in the above example (IID_IShellFolder). This is the most important interface of ShellDesktop.

The following partial code shows the call for `CoCreateInstance()` in a greater context. This partial listing generates a `ShellLink` object. Then, using the `ShellLink` object, a shortcut is created to the `AUTOEXEC.BAT` file on the Desktop. The basic processes in accessing OLE objects and not the individual details are emphasized.

```
#include <windows.h>
#include <shlobj.h>

int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine,
                   int nCmdShow )
{
    typedef IShellLink *LPSHELLLINK;
    typedef IPersistFile *LPPERSISTFILE;
    LPSHELLLINK pShellLink;

    CoInitialize( NULL ); // initialize OLE

    //generate ShellLink object with pointer to IShellLink interface
    if( SUCCEEDED( CoCreateInstance( &CLSID_ShellLink, NULL,
                                     CLSCTX_INPROC_SERVER, &IID_IShellLink, &pShellLink ) ) )
    {
        LPPERSISTFILE pPersistFile;

        // generate pointer to IPersistFile interface in the ShellLink object
        if( SUCCEEDED( pShellLink->lpVtbl->QueryInterface( pShellLink,
                                                         &IID_IPersistFile, &pPersistFile ) ) )
        {
            WORD wsz[ MAX_PATH ];
            HRESULT hr;

            // Set Shell-Link to refer to AUTOEXE.BAT
            hr = pShellLink->lpVtbl->SetPath(pShellLink, "c:\\\\AutoExec.Bat");
            hr = pShellLink->lpVtbl->SetDescription(pShellLink, "Autoexec");

            // Save ShellLink object and release IPersistFile
            MultiByteToWideChar( CP_ACP, 0, "c:\\\\win95\\desktop\\autoexec.lnk",
                                -1, wsz, MAX_PATH );
            hr = pPersistFile->lpVtbl->Save( pPersistFile, wsz, TRUE );
            hr = pPersistFile->lpVtbl->Release( pPersistFile);
        }
        pShellLink->lpVtbl->Release( pShellLink );
    }
    CoUninitialize(); // switch off/disable OLE for this application
    return 0;
}
```

UNICODE conversion

One of the most basic parameters determining the compatibility between two computer systems is the character set that is used. Accurate communication is impossible when one system interprets an X as a U. Conversion functions can help if the differences between the two systems are known. However, conversion functions also mean more time and work for the developer.

Although Windows 95 works largely with the uniform ANSI character set, you're not completely spared from conversion problems. One example of this is the Shell. After all, part of the Shell's job is to access objects that could come from computers with other character systems (such as Windows/NT, which works with the UNICODE character set). The UNICODE character set represents all the languages in the world. The main difference from ANSI code: Each character of a string no longer consists of one byte, but instead, is made up of 2 bytes. While this helps international relations, it also guarantees that all of your fine string handling routines will break down. We don't even want to talk about what happens to screen output.

Windows 95 doesn't use UNICODE, which might not help international relations, but it does improve how the system runs on a PC with 4 or 8 Meg of RAM. If each string requires twice as much disk space, merely because it appears in UNICODE, your computer memory will fill up fast.

However, at the boundary between your program code and the software interface of the Desktop, you may not be able to avoid UNICODE. Therefore, strings must be converted between Unicode and the conventional ANSI character set on a regular basis. Naturally, some characters are lost in the process (how else do you make 1 byte out of 2), but all UNICODE strings are converted perfectly with the Roman alphabet.

The two functions for converting from ANSI to UNICODE and back are called MultiByteToWideChar() and WideCharToMultiByte(). They're included in the KERNEL32.DLL and defined in the WINNLS.H Include file. The names for the functions are a bit misleading, since neither ANSI nor UNICODE appear in the names. WideChar means UNICODE and MultiByte refers to ANSI. That is, most of the time. Instead of ANSI, two other 1 byte character sets could serve as the destination or starting point of the conversion: The standard DOS character set (referred to as the OEM character set) and the Macintosh character set. That's why it's not called AnsiToWideChar(), but rather, has the general name of MultiByteToWideChar().

First, we'll look at the WideCharToMultiByte() function for converting received UNICODE strings to ANSI strings.

```
int WideCharToMultiByte(    // converts UNICODE string to ANSI
    UINT    CodePage,      // MultiByte character set (ANSI, DOS, Mac)
    DWORD    dwFlags,      // 0
    LPCWSTR  lpWideCharStr, // Address of UNICODE string in memory
    int      cchWideChar,   // length of the UNICODE string, -1 for automatic
                          // detection
    LPSTR     lpMultiByteStr, // Address of buffer for the converted string
    int      cchMultiByte,   // size of the MultiByte buffer in bytes
    LPCSTR    lpDefaultChar, // character for non-converted UNICODE
    LPBOOL    lpUsedDefaultChar // pointer to flag indicating non-converted
                          // UNICODE
);
```

In the first parameter, a predefined constant must be specified first for the character set into which the UNICODE string is to be converted: CP_ACP represents ANSI, CP_MACCP represents the character set of the Macintosh and CP_OEMCP represents the DOS-ASCII character set. The letters "cp" appear so often because CP is the abbreviation for "codepage", which means the same as character set. As the second function parameter, a flag is expected, which dictates the procedure with non-convertible UNICODE characters. For the contexts we're dealing with, you can always specify 0 for this parameter.

The next two parameters lpWideCharStr and cchWideChar describe the UNICODE string to be converted. lpWideCharStr describes its address in memory and cchWideChar describes the number of characters in the string. If you don't know the number and can assume that the UNICODE string is null-terminated (that is, the string ends with the UNICODE NUL character), simply specify a -1 here. The function then determines the length of the UNICODE string itself.

With the next parameters, lpMultiByteStr and cchMultiByte, the caller determines the location of the converted string. lpMultiByteStr handles the address of the string, while cchMultiByte has to do with the number of characters that fit in the address. The function uses this information to make sure that there is enough room in the buffer.

Finally, with `lpDefaultChar` and `lpUsedDefaultChar` the function expects two parameters that define the procedure regarding nonconvertible UNICODE characters. All UNICODE characters that don't have any equivalents in the selected 1 byte character set are nonconvertible. The caller points to a CHAR in memory containing the placeholder for the nonconvertible character using `lpDefaultChar`. If you specify NULL, the function will use the system's default placeholder character. If you want to know whether at least one character could not be converted, specify the address of a BOOL variable in the last parameter, `lpUsedDefaultChar`, instead of NULL. The function loads the address with TRUE when its executed if at least one UNICODE character could not be converted. Otherwise, the function loads the address with FALSE.

There's one special feature concerning the `cchMultiByte` parameter we still need to mention. It specifies the length of the conversion buffer. If you don't know the length of this buffer because you're uncertain of how many characters the UNICODE string contains, specify 0 when calling `WideCharToMultiByte()`. The function won't perform the conversion. It instead returns the number of characters that would result from a conversion. Therefore, in this case you can specify NULL because there won't be any conversion.

Then you allocate memory for the converted string using the dynamic memory and pass a pointer to it the next time you call the function, however, this time specifying the number you discovered, so that the conversion actually takes place. Remember the buffer requires one more byte than the number of characters that were returned so that the NUL byte will also fit. In this case (`cchMultiByte > 0`) the function result specifies the number of converted characters in the buffer. If you get a function result of 0, the UNICODE was either empty, or an error occurred. The following code sequence illustrates the procedure for converting UNICODE strings whose length is not known.

```

/*****
/* UnicodeToAnsi : Converts Unicode strings to Ansi strings          */
/*-----*/
/* Parameter :    lpUnicode - Address of Unicode string             */
/* Return value : Address of ANSI string or NULL, if error           */
/*-----*/
/* Info : The caller must deallocate the returned string using free().*/
*****/
LPSTR UnicodeToAnsi( LPWSTR lpUnicode )
{
    LPSTR lpAnsi;
    int cch;

    // Get number of characters in the Unicode string -----
    cch = WideCharToMultiByte( CP_ACP,
                              0,
                              lpUnicode,
                              -1,
                              NULL,
                              0,
                              NULL,
                              NULL ) + 1;

    // Allocate memory -----
    lpAnsi = malloc( cch );
    if( lpAnsi ) // Copy wide characters to lpAnsi -----
        WideCharToMultiByte( CP_ACP,
                              0,
                              lpUnicode,
                              -1,
                              lpAnsi,
                              cch,
                              NULL,
                              FALSE );
}

```

```

        NULL,
        NULL );

    return lpAnsi;
}

```

In the opposite direction, the `MultiByteToWideChar()` function converts strings from the ANSI, PC or MAC character set to UNICODE strings.

```

int MultiByteToWideChar(           // converts ANSI string to UNICODE
    UINT      CodePage,           // CP_ACP, CP_MACCP or CP_OEMCP
    DWORD     dwFlags,             // 0
    LPCSTR     lpMultiByteStr,     // address of string to be converted
    int        cchMultiByte,       // length of the string or -1 for automatic
                                   // detection
    LPWSTR     lpWideCharStr,      // pointer to buffer for UNICODE string
    int        cchWideChar         // size of the buffer
);

```

The function works similar to its counterpart regarding the parameters. The main difference is that you won't need any information in case the characters cannot be converted to the UNICODE character set. This eventuality is out of the question, because the UNICODE character set is a combination of all three character sets that can be converted.

If you enter -1 instead of the length of the string for `cchMultiByte`, the function calculates the length of the string on its own, assuming that it is a normal C string with a terminating NUL byte. With regard to `cchWideChar`, you can use the same trick that you used with `WideCharToMultiByte()`. If you specify 0 with the call, the function automatically returns the characters required for the UNICODE string (excluding the terminating NUL byte).

```

/*****
/* AnsiToUnicode : Converts Ansi strings to Unicode strings          */
/*-----*/
/* Parameter :    lpAnsi - Address of Ansi strings                  */
/* Return value : Address of Unicode string, or NULL in case of error */
/*-----*/
/* Info : The returned string must be deallocated by the caller again */
/*         using free().                                              */
*****/
LPWSTR AnsiToUnicode( LPSTR lpAnsi )
{
    LPWSTR lpUnicode;
    int cch;

    // Get number of characters in Ansi string -----
    cch = MultiByteToWideChar( CP_ACP,
                              0,
                              lpAnsi,
                              -1,
                              NULL,
                              0 ) + 1;

    // Allocate memory -----
    lpUnicode = malloc( cch * 2 );
    if( lpUnicode ) // Copy wide characters to lpsz -----
        MultiByteToWideChar( CP_ACP, // Ansi characters to lpUnicode ---

```

```

        0,
        lpAnsi,
        -1,
        lpUnicode,
        cch * 2 );

    return lpUnicode;
}

```

The IMalloc interface

Although it's not part of the Shell, you'll frequently come into contact with the IMalloc interface in OLE programming. Its name suggests that it has to do with managing dynamic memory at runtime. Instead of the normal Heap functions, all OLE interfaces use this interface internally to allocate dynamic memory. If you get a memory object from an interface function, its memory was allocated using IMalloc, so it must also be released by this interface. That's the point at which you come into contact with IMalloc within a Desktop application. As a rule, the interface won't take back memory returned in this manner, you have to release it yourself, using the IMalloc interface. However, you can manage custom memory areas using malloc() and free(), even those that were passed to an interface method using pointers.

Besides the methods of the IUnknown interface, IMalloc contains the following methods:

Methods of IMalloc	
Method	Task
Alloc	Allocates memory
Free	Releases allocated memory
Realloc	Changes the size of an allocated memory area
GetSize	Supplies the size of an allocated memory area
DidAlloc	Checks whether a specific memory block was allocated with the existing instance of IMalloc
HeapMinimize	Releases unused memory on the Imalloc heap back to the system

Use the OLE API CoGetMalloc() function to get a pointer to an IMalloc object. It lets you call methods like Alloc() and Free():

```

HRESULT CoGetMalloc(
    DWORD          dwMemContext,           // reserved, always specify 1
    LPMALLOC FAR*  ppMalloc)             // pointer to Var which receives the interface
                                         // pointer
);

```

Because OLE clients are often confronted with the problem of using IMalloc(), only because their OLE server used it to allocate returned memory, the most important IMalloc methods are mapped to three API functions from the OLE API: CoTaskMemAlloc(), CoTaskMemFree() and CoTaskMemRealloc(). These functions make it a bit easier for the caller, because you don't have to get a pointer to an IMalloc object using CoGetMalloc() first. So, we use these functions in the programming examples in the Desktop chapter/section. CoTaskMemFree() is usually used because you want to release memory that was allocated by an OLE object using the call for IMalloc::Alloc.

```

LPVOID CoTaskMemAlloc(    // allocate memory using IMalloc::Alloc
    ULONG  cb             // size of desired buffer in bytes
);                        // returns pointer to buffer or NULL

LPVOID CoTaskMemRealloc(  // Increase/decrease memory using IMalloc::Realloc
    LPVOID pv,           // pointer to allocated block
    ULONG  cb            // new size in bytes
);                        // returns pointer to a block or NULL

```



```
void CoTaskMemFree(                // Release memory using IMalloc:Free
    LPVOID pv                      // Pointer to allocated buffer
);
```

The IPersistFile interface

You will also encounter the IPersistFile interface in Desktop programming. This is an OLE defined interface for loading and saving objects in files. The interface is not implemented but only defined by the OLE subsystem. It's implemented by many objects, which furnish a standardized mechanism for loading and saving the objects on the instructions of a client.

Anyone who has an object instance, regardless of the object type, can use its IUnknown-Interface to find out whether the object provides the IPersistFile interface. Many objects, including the objects of the Desktop, don't contain methods for loading and saving in their individual interfaces but implement the IPersistFile interface instead for this functionality.

If IPersistFile is available, the object offers the client the option of saving itself in a file or of loading itself from the file. The client doesn't have to perform this task and worry about the size and structure of the object but can use the standardized interface with its simple Load and Save methods. A call for QueryInterface() is enough to determine whether this is possible. All you need for this is the type definition of IPersistFile, which is part of OLE2.H. OLE2.H also has the externally available variable IID_IPersistFile, which you can use to get the interface ID of IPersistFile, and whose address you pass to QueryInterface().

Methods of IPersistFile	
Method	Task
GetClassID	Supplies the class ID of the object whose IPersistFile interface you are using.
IsDirty	Checks whether an object needs to be saved.
Load	Loads object from file.
Save	Saves object in file
SaveComplete	Indicates completion of a save.
GetCurFile	Supplies the complete path to the current object.

IPersistFile contains six methods. Only two of these methods are required: Load and Save. This is also the case in our example, which is why we are emphasizing both of these methods.

Load is used to load an object saved in a file into memory. An object instance that can receive the data must already exist in memory. (You couldn't call the interface without this object instance anyway). Besides the mandatory this pointer, the address of a C string with complete path and filename is expected. The file will normally have been saved there by a previous client using the Save method of the IPersistFile interface.

```
HRESULT Load (                // loads object from file
    IPersistFile *This,
    LPCOLESTR pszFileName,    // pointer to string with filename
    DWORD dwMode              // access mode, STGM_
);
```

Specify access to the file using the dwMode parameter. If you don't want to grapple with the STGM constants defined for it, simply specify 0, that will work. Otherwise, the following constants are available,

STGM constants for IPersistFile::Load	
Constants	Task
STGM_READ	File can be read
STGM_WRITE	File can be written
STGM_READWRITE	Read and write is allowed
STGM_SHARE_DENY_NONE	Other callers are allowed to access the file simultaneously
STGM_SHARE_DENY_READ	Other callers are not permitted to read the file simultaneously
STGM_SHARE_DENY_WRITE	Other callers are not permitted to write to the file
STGM_SHARE_DENY_EXCLUSIVE	Warns all other callers to keep away

You get a function result of S_OK if the file was loaded as desired, E_NOMEMORY if the call failed due to insufficient memory and E_FAIL if something else was responsible. For example, you would get this last function result if the file could not be found, the object detected that it wasn't the same type or the file was saved under another version of the object.

The Save method also expects a pointer to a UNICODE string with the path and filename under which the file is to be saved after the this pointer. Specifying NULL saves the file under the same name under which it was loaded or saved the last time.

```
HRESULT Save (                                     // saves object in file
    IPersistFile *This,
    LPCOLESTR    pszFileName,    // pointer to string with path and filename
    BOOL         fRemember      // Save or Save As
);
```

The fRemember parameter decides whether the object remembers the new filename or whether you perform a kind of "Save as" operation, in which only a copy is created with a different filename. If you specify TRUE, the new filename replaces the old one. FALSE, on the other hand, sees to it that the file is only saved under a different name, you continue working on the present file. You can determine which filename is current, i.e., where the file is located, using the GetCurFile method. You must pass the address of a variable to this method, which loads it with the pointer to a buffer allocated by the method. In this buffer you will find the current path and filename as a C string. The client has to release this buffer before its termination. To release the buffer, use either IMalloc::Free or CoTaskMemFree().

```
HRESULT GetCurFile (                             // returns current filename
    IPersistFile *This,
    LPOLESTR     *ppszFileName    // pointer to Var that receives Ptr to
                                   // string
);
```

The following summarizes the other IPersistFile methods.

```
HRESULT GetClassID (                             // furnishes object identity
    IPersistFile *This,
    CLSID        *pClassID    // pointer to Var that is to receive the class ID
);
HRESULT IsDirty (                                 // indicates whether changes were saved
    IPersistFile *This
);
HRESULT SaveCompleted (                          // indicates completion of a Save
    IPersistFile *This,
    LPCOLESTR    pszFileName    // pointer to string with path and filename
);
```

You'll find an application example for calling IPersistFile methods in Chapter kkk on creating shortcuts.

43

Working With The Shell

This chapter highlights the Windows 95 Shell and the items that represent it. These items include, most importantly, the Desktop with its folders, programs, shortcuts and, naturally, the Explorer. The purpose of a shell is ultimately to let users access the resources of a computer system. This also includes the hard drives and CD-ROM drives as well as a network environment or system configuration. Shells act as a go-between for the user and the actual system functionality, as represented by the operating system kernel with its extensions. Therefore, the Shell is not part of the operating system, but instead, is run as an independent, although privileged program. It's possible to exchange shells if the manufacturer of the operating system doesn't place any special obstacles in the way. This is customary in UNIX environments, for example, where several shell versions circulate with reasonable success.

The Windows 95 Shell also fits within this idea at a very advanced level. However, if you removed the highly developed components of the user interface from the Desktop, such as the icons, the trees, lists and context menus, what remains is the same thing the DOS Shell and its command line. In other words, what remains is the opportunity to create, copy, move files and directories. Above all, a shell enables you to start programs. In other words, if there are no shell, there are no program calls. Any programmer who has experimented with programming without a shell has discovered how difficult life can get on a PC.

This chapter discusses the various services that the Shell and Desktop have to offer programs. Traditional services and operations for the Shell include:

- Copying
- Deleting
- Renaming files
- Starting programs

The Shell, however, also does the following:

- Managing program groups
- Creating shortcuts
- Printer control
- Saving the last addressed documents
- Link between file extensions
- Document types and servers
- Displaying dialog boxes for loading and saving

Two methods of performing the different services are possible. One is from the Shell API, which is in a file called SHELL32.DLL. You add the SHELLAPI.H include file at C level to get the declarations of the required functions, types, constants and structures. Among these are some functions that provide access to the desktop and its controls. However, most desktop functionality is exported through two OLE interfaces: IShellFolder and IShellLink. You will find their declarations in a file called SHLOBJ.H.

Indeed, it appears that the basic functionality of the whole had been implemented as OLE objects. Then someone had the idea perhaps that was too difficult for all the developers who had never worked with OLE. So they added some API functions, which only serve as a kind of wrapper for OLE calls. However, you can't do everything with the API functions that you can from the OLE interface. Therefore, things are a bit chaotic at system level. Furthermore, different items seemed to have been thrown together in the Desktop. For example the following all exist partially as files, directories or programs on disk. However, they also exist partially as virtual constructions in the Shell Namespace.

- Files
- Folders
- Program groups
- The Network Neighborhood
- The Control Panel
- The Recycle Bin

The Shell Namespace and the Desktop are two separate subjects and are what this section discusses.

The Shell Namespace

The Shell Namespace is perhaps the most exciting item in this chapter. However, you won't necessarily find it often in concrete programming. You'll appreciate the Shell API if you only copy files and start programs or are interested only in displaying the dialog boxes for loading and saving files. This is true even without the OLE interface.

However, the most interesting thing about the Desktop is how it manages entirely different items according to a uniform system. It's not by pure chance that OLE interfaces were used for this purpose, because object oriented technology - OLE is only one example of it - is the key to this technology. The deciding catchword is *polymorphism*. This is the option of treating different types of items (objects) similarly with one piece of code without worrying about the different characters of the objects. The ShellFolder object embodies this idea with its IShellFolder interface. You'll use these interface to access the Shell Namespace. You'll need at least a basic understanding of OLE technology to understand the processes and services offered by the Desktop and the Shell.

A word about filenames

Before we start, let's talk about long filenames for a moment. The Desktop may at first seem to be the very embodiment of long filenames. However, you'll hardly come across changes in this area in Shell and Desktop programming. The only change you will notice is that filenames can now be much longer, namely up to 255 characters. So, the corresponding buffers no longer require space for only 64 characters. They must, instead, have enough space for MAX_PATH characters. MAX_PATH is the name of a C constant defined for this purpose. You usually only pass along filenames along. For example, you get filenames from the user to pass them on to a system function (or vice versa). Therefore, you don't need to consider the distinction between long and short filenames. This also applies for UNC names, which start with a double slash and specify a server name (\\SuperServer\\MITI1\\Hello).

The Shell Namespace

Everything you see on the Desktop belongs to the Shell Namespace. This follows a hierarchy of objects, most of which are folders. Each object, within this Namespace, whether file, folder or program group, gets a unique name. This name distinguishes it from all the other objects. You must be able to tell the objects apart and address them on an individual basis.

Structure of the Namespace

The structure of this Namespace follows the structure of a drive with root directory, subdirectories, sub-subdirectories, etc. The only difference is that you aren't necessarily working with real directories with the Shell Namespace. Instead, you're working with different types of objects. Although the hierarchical arrangement is similar to a file system, not everything that appears in the Shell Namespace must be a subdirectory or file on a storage medium.

Display the Namespace easily by starting the Windows Explorer. The Desktop is at the root of the Namespace. The Desktop, like all folders, is represented internally by a ShellFolder object. You might say the Desktop is the root directory of the Namespace. All the items within the Namespace that contain other objects are ShellFolder objects. This includes the folders that you create on the Desktop, the Start menu with its entries and, of course, the directories. This is shown in the display of drives in the Explorer. Regardless of their characteristic features, they're all ShellFolder objects internally.

The Namespace doesn't have only ShellFolder objects by themselves, which are only containers for other objects. The actual contents also have to be somewhere. The contents are created by files and programs. These files and programs, called *file objects*, are located in the different ShellFolder objects. These are the elements on which everything is based. You cannot execute containers. You can only execute the elements contained within.

While the Shell Namespace is built when you start Windows 95, it is extended and maintained at runtime. For example, the Shell might discover at startup that the computer is connected to several servers. However, it doesn't have to query them for their contents and display them in the Namespace if the user doesn't attempt to get a look at the servers. If the user moves while browsing the Namespace to an object whose contents haven't yet been read, the Windows Explorer/Shell expands the Shell Namespace below the ShellFolder object accordingly. If you're working with network drives, it may take some time for the directory tree to be loaded. You'll notice this because the Explorer temporarily seems to freeze its display.

A significant part of the Namespace is determined by the illustration of drives, their directories and the files the directories contain. Somewhere within the Namespace is a ShellFolder object for each drive and each server. The Shell considers this ShellFolder object to be the starting point for displaying the drive in the Namespace. In a sense, this ShellFolder object is the root directory. Corresponding objects are placed within this ShellFolder object for all of its files and directories. The subdirectories are ShellFolder objects, while the files are FileObjects. This also applies to all subdirectories, their subdirectories and the files contained in the subdirectories. Each subdirectory gets its folder within the Namespace. This folder is in the ShellFolder object representing the parent directory.

If you know the starting point of a drive within the Namespace, you can run through the objects contained within the drive. You can then get a picture of the structure of the drive and its contents. While you can do the same thing with the normal API functions for finding files and directories, one difference is important. You can browse through the represented drives and their contents with the OLE interfaces of the Shell. Furthermore, you can browse through the other elements in the Shell Namespace. These elements include the entries in the Start menu or the contents of the Control Panel. This is precisely the point of the standardization that the Shell in the Desktop is executing. In contrast, all you can do with the API functions is query files and directories.

A series of ShellFolder objects in the Shell Namespace doesn't represent any real or existing elements. This is unlike the ShellFolder objects, which represent a drive's directories and files. Instead, the ShellFolder objects represent virtual elements that the Shell inserted into the Namespace of its own accord. Some examples are the Recycle Bin or the collection of installed fonts. We'll come back to this topic a bit later.

Naming ShellFolder Objects

If you want to work with or edit an item within the Namespace using API functions or one of the methods of the IShellFolder interface, you must be able to name it. Then the function knows what it's supposed to do to which item.

In this respect, the Shell follows the idea of the file system. Items within the Namespace are called by following their path from the root of the system (the Desktop) through all the ShellFolder objects that you must open to get to the desired item. The names of the various ShellFolder objects on this path are arranged in sequence (as you do when specifying an absolute path for filenames). Alternatively, you could also specify a relative path if you define the starting point at the same time. For example, you could say you are starting at the ShellFolder object that refers to the C: root directory. All you must do is name the path with the ShellFolder objects that start below this starting point.

All the API functions of the Shell that act on an item of the Namespace expect an absolute path. This path starts with the Desktop. The methods of the IShellFolder interface expect relative specifications regarding a previous ShellFolder object. We'll show you how this works later in the chapter.

One important difference to naming files and directories in the framework of the file system is that you don't use ASCII strings to name ShellFolder objects. Use item identifiers, which are binary structures, instead. You don't need to be familiar with their structure because you always pass them to methods of the IShellLink interface or an API function for editing. Item identifiers are a tribute to the idea of object oriented technology. It's possible there will be items in such a Namespace that cannot be uniquely described simply based on their name. An item identifier usually contains a name, which is hidden next to other information that describes the object uniquely.

To build a path using item identifiers, for example, to describe the location of an object compared to the root of the Namespace (i.e., an absolute path), link the appropriate item identifiers in memory. For instance, to address the Accessories program group, the appropriate path includes four such item identifiers. These are called item IDs. First is the item ID for the Desktop object, which is the root of the Namespace. This is followed by the item identifier for the Start menu. Next is the item identifier for Programs and finally the item ID for Accessories. In practice, however, you ignore the item ID for the Desktop. This is already implicitly defined as the starting point with an absolute path. So the required list of item identifiers for accessing the Accessories folder consists of three, not four item identifiers. If you wished to address an item from Accessories, simply add the appropriate item identifier of the object to this chain.

This chain is referred to as an item identifier list, or IDL for short. Compared to the methods of the IShellLink interface and the Shell API, such a list is referenced by a pointer called PIDL. Whenever we talk about a PIDL in this chapter, we also mean

a corresponding item identifier list to which the PIDL refers. Although you don't know the exact structure of the individual item identifiers, you can make your way from level to level to get to specific item identifiers. At the beginning of each identifier is its length as a USHORT, or a word. Simply add the contents of this word to its address to get a pointer to the next item identifier within an IDL. The end of a list is marked by an item identifier with a length of 0. There is no more data after this length specification.

An item identifier is represented within the SHJLBJ.H include file by the SHITEMID type. A pointer to the identifier is represented by LPSHITEMID. Use the cb field to get the length. CB represents "count byte".

```
typedef struct {                                // an item identifier (MKID)
    USHORT cb;                                // size of the item identifier including this field
    BYTEabID[1];                             // binary representation of variable length
} SHITEMID, * LPSHITEMID;
typedef const SHITEMID * LPCSHITEMID;
```

The pointer to a list of item identifiers is defined as the LPITEMIDLIST type. It refers to a structure with a single item of the SHITEMID type. This item is called mkid, the abbreviation for an item identifier (mkid = make item identifier).

```
typedef struct {                                // Item Identifier List (IDL)
    SHITEMID mkid;                             // the actual Item Identifier
} ITEMIDLIST, * LPITEMIDLIST;                 // LPITEMIDLIST is a PIDL
typedef const ITEMIDLIST * LPCITEMIDLIST;
```

If you get a PIDL from a Shell API function or one of the IShellFolder methods, the memory for it usually comes from the IMalloc interface. That's why you must free this memory again before the end of the program. Use either the Free method from IMalloc or the OLE API CoTaskMemFree() function to free this memory. Because item identifiers are binary structures, which have no meaning for the user, each ShellFolder object contains a "display name". This is the name the user sees when they look at the object in Windows Explorer or on the Desktop. With objects that create a concrete item on a mass storage system, it is identical to the file or directory name. With virtual ShellFolder objects, the display name is chosen artificially. The following sections show how to set or query display names from a program.

Help functions for working with PIDLs

As some basic functions are required for editing strings, the same operations also appear repeatedly with PIDLs. For example, such operations include combining several item identifiers into an item identifier list or browsing through such a list. Therefore, we prepared a small help module with some of these functions.

The Shell API

The easiest way to contact the Shell is to use the API functions from SHELL32.DLL. However, don't expect too much from these functions. They won't, for example, get you into the Shell's Namespace. At any rate, you'll need some of these functions to work with the Shell and its OLE interfaces. The following table summarizes the functions.

API Functions of the Shell	
Function	Task
SHGetSpecialFolderLocation	Returns an Item Identifier list for one of the virtual Desktop folders.
SHGetPathFormIDLlist	Converts an Item Identifier list into a path and filename.
SHGetDesktopFolder	Returns the interface to the Shell object representing the Desktop as the root of the Namespace.
SHAddToRecentDocs	Adds a document to the list of recently opened documents within the Start menu.

adjust table

API Functions of the Shell (continued)	
Function	Task
SHBrowseForFolder	Displays a dialog box that lets users navigate through the Namespace and select folders.
SHChangeNotify	Informs the Shell of changes in the file system that could affect the Namespace of the Shell.
SHFileOperation	Performs a file operation (copy, move, rename or delete files).
SHGetFileInfo	Returns information about a Shell object
SHGetMalloc	Returns the IMalloc interface of the Shell.

Origin of the Desktop items

Although you cannot access the Shell Namespace directly through the API functions, they still function partially as an interface between the OLE interfaces and the rest of the API system. PIDs play the main role here. They allow users to name an object in the Shell Namespace. SHGetSpecialFolderLocation() represents a function that helps you determine the PIDL for an object in the Namespace. For example, you could use this in a call to SHBrowseForFolder(). This is a powerful function. It lets you use a dialog box from a program that enables users to navigate through areas of the Shell Namespace and select folders. First, the following is SHGetSpecialFolderLocation().

```

HRESULT SHGetSpecialFolderLocation(    // returns PIDL for a virtual folder
    HWND          hwndOwner,          // window handle of parent
    int            nFolder,            // CSIDL constant for desired folder
    LPITEMIDLIST *ppidl               // pointer to var that receives PIDL
);

```

In any event, the function expects the window handle of the caller as the first argument. This is done in case it's necessary to display dialog boxes. Specify NULL if no window handle is available. The second parameter decides to which folder you direct the function call. The CSIDL constants from the following table represent the various Desktop items. Use this function to determine their PIDL.

CSIDL constants for SHGetSpecialFolderLocation()	
Constant	Represents
CSIDL_DESKTOP	The Desktop folder, the root of the Namespace
CSIDL_PROGRAMS	The system directory in which the individual programs are stored as subdirectories
CSIDL_CONTROLS	The folder containing the programs of the Control Panel
CSIDL_PRINTERS	The folder containing the installed printers
CSIDL_STARTUP	The system directory containing the files of the Autostart program group
CSIDL_RECENT	System directory in which shortcuts to the most recent edited document are placed
CSIDL_SENDTO	The system directory containing the files to be sent
CSIDL_BITBUCKET	System directory where files are copied that are going to the bit bucket
CSIDL_STARTMENU	The system directory of the Start menu
CSIDL_DESKTOPDIRECTORY	The system directory in which the items placed on the Desktop are stored
CSIDL_DRIVES	The folder containing the items of "My Computer"
CSIDL_NETWORK	The folder at the root of the network hierarchy
CSIDL_NETHOOD	The system directory with the items of the Network Neighborhood
CSIDL_FONTS	The folder containing the installed fonts
CSIDL_TEMPLATES	The system directory for reusable templates as random files

These are predominantly virtual folders. The Shell artificially creates these folders to combine all the resources of the computer in a manageable frame. Remember, we've been talking about both folders and system directories. The difference between the two will be explained later in this chapter.

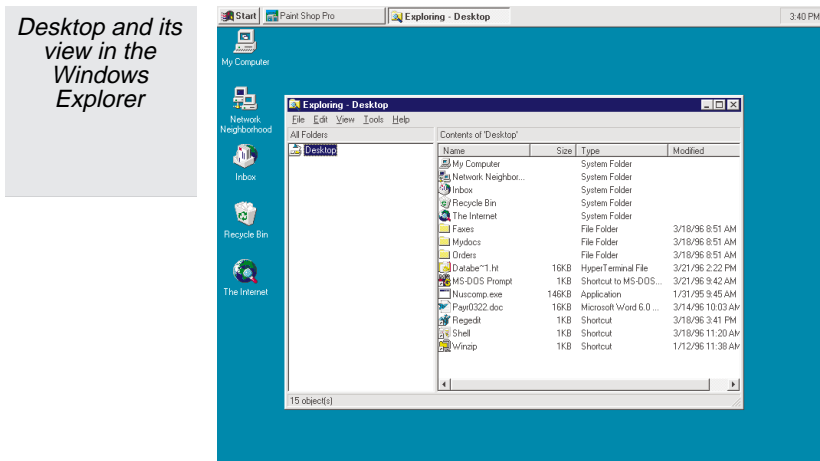
Along with one of these constants, `SHGetSpecialFolderLocation()` expects the address of a `LPITEMIDLIST` type variable as the last parameter. If successful, the function loads it with `PIDL`, that is, with the address of the item identifier list of the desired object. You can tell whether the function call was successful by the return value of the `HRESULT` type. `HRESULT` is a data type used primarily by OLE functions. There are two predefined macros you can use to check this value, called `ERROR` and `SUCCEEDED`. You will find an example for calling and working with this function in the section on the `SHBrowseForFolder()` function.

Folder or Directory

Let's turn our attention back to the various `CSIDL` constants. Some of these constants concern folders while other constants concern directories. Again, this highlights the relationship between folders and directories. On the one hand, the Shell evidently maps the structure of a drive with its files and directories directly to a part of the Namespace tree. However, the Shell also uses parts of the file system to place its virtual objects in persistent storage. These virtual objects include the program groups and their subgroups as well as the contents of the Desktop. What appears here in the Shell Namespace simply represents the mapping of specific directories in which the Shell stores program groups and the contents of the Desktop.

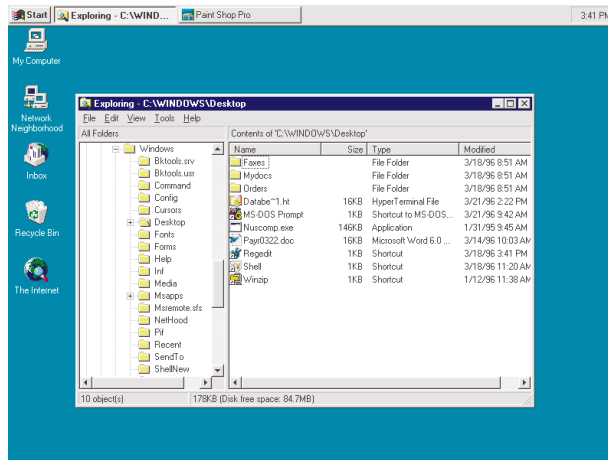
For example, the folders and shortcuts on the Desktop must be stored together with their contents somewhere on disk. Otherwise this would all be lost when you switched off the computer. The Desktop requires a physical storage place, and the file system lends itself admirably to the cause. What you see on the Desktop is usually simply the contents of a special directory. If you move a file from a hard drive directory onto the Desktop, the file is simply moved from its previous directory to the directory that the Desktop uses for storing its contents. What appears to be the Desktop to the user is, in reality, simply the contents of a hard drive directory reserved for the Shell.

The following illustration shows a Desktop and its view in the Windows Explorer on the Desktop. You can see the icons of MyComputer, the Control Panel, the Network Neighborhood, the Windows Explorer, the Inbox, the Recycle Bin, and several folders.



The following illustration looks very similar at first. However, the difference is in the details. Windows Explorer no longer displays the contents of the Desktop but rather of a directory on the hard drive called `C:\WINDOWS\DESKTOP`. Apart from two exceptions, what you see here is what you see on the Desktop because this directory is the Desktop. The Shell places all the objects that appear on the Desktop in the `DESKTOP` subdirectory of the Windows directory. The folders of the Desktop are subdirectories (or shortcuts to subdirectories in another part of the file system). The icons appearing on the Desktop are programs or shortcuts to these programs.

View of the Desktop folder



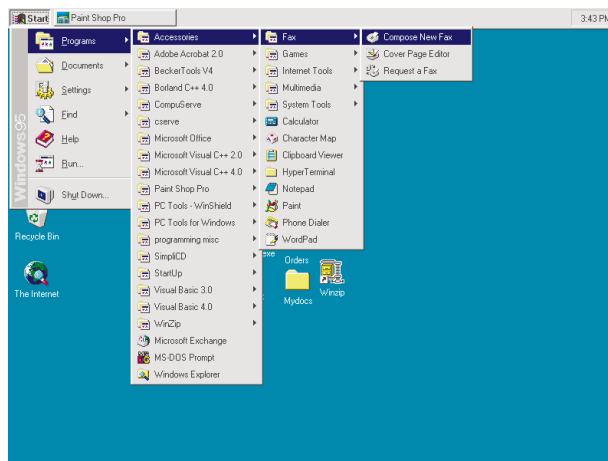
There's an easy way to check. Use the Explorer to create a few new directories or files in the Desktop subdirectory of your Windows 95 directory (whatever its name is). Almost immediately, these items will also appear on the Desktop. The same is true if you use the DOS Prompt to create the directories or to copy files to this directory. The file or directory name appears as an object name on the Desktop. It's that easy!

So, creating new Desktop folders and items for a program leads to generating new subdirectories, creating corresponding shortcuts or copying files to the directory that the Desktop reserves for its storage. Determine this directory by specifying the CSIDL_DESKTOPDIRECTORY constant when you call SHGetSpecialFolderLocation(). However, you only get back a PIDL. It must first be converted to a proper path name to give you the location of the desired directory. We'll show you how to do this. First, let's have a look at the other virtual Desktop items and how they are stored on the hard drive.

For the most part, as with the Desktop, it's a matter of subdirectories of the Windows 95 directory. For example, there's Start for the Start menu, Start\Programs for the programs in the Start menu or Fonts for the fonts available to the system. Any subfolders the item might have are simply generated as directories in the hard drive directory. Both of the following illustrations show how the program groups and their programs are stored in this manner.

The first illustration shows the expanded Start menu with the Programs menu and its program groups. The Fax program group is open so three programs appear in this group.

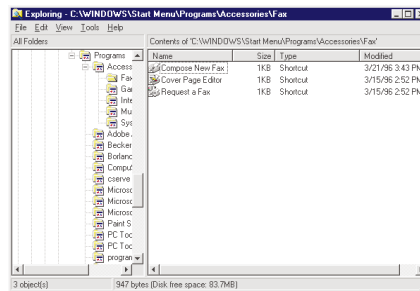
Expanded Start menu



A look at the Windows 95 directory reveals how the program groups get on the Desktop. The Windows 95 directory has a subdirectory called StartMenu, containing items of the Start menu as files or directories. There is, among others, the Programs directory. It contains the program groups and some files that can be displayed and called from this menu. Some program groups have subgroups, such as Accessories. It has the Fax subdirectory. This subdirectory contains shortcuts to three programs that you can call from the Desktop.

It becomes obvious a program can easily create new program groups and program entries by generating the corresponding directories and placing the program files or shortcuts to these files. Here, too, you must first determine the directory, which is why you specify the CSIDL_PROGRAMS constant when you call SHGetSpecialFolderLocation().

*The same
hierarchy in the
Explorer*



The table on the right summarizes the other Desktop items and the default directories in which they are placed.

However, don't put too much stock in the directories listed here. Situations can occur where the Shell will choose other directories for specific items. An example of this is when the Desktop has been set up for several different users. Then there will be different Desktop directories because each user sets up their folders and programs on their Desktop. In such a case, SHGetSpecialFolderLocation() gives you those directories that are valid for the user currently logged on.

Desktop items and the default directories in which they are stored on the hard drive	
Constant	Default Directory (C:\WINDOWS\...)
CSIDL_DESKTOPDIRECTORY	Desktop
CSIDL_STARTMENU	Start Menu
CSIDL_PROGRAMS	Start Menu\Programs
CSIDL_STARTUP	Start Menu\Programs\Startup
CSIDL_FONTS	Fonts
CSIDL_NETHOOD	NetHood
CSIDL_RECENT	Recent
CSIDL_SENDTO	SendTo
CSIDL_TEMPLATES	ShellNew

Other items, such as the Control Panel or the folder with the installed printers, receive their contents from the Registry. The Shell notes the names of the installed printers as well as their drivers. The Shell notes the programs for the Control Panel that represent the its different components. By using this information, the Shell builds a virtual folder in the Namespace containing the corresponding items. It doesn't, however, place the items in a specific hard drive directory.

SHGetPathFromIDList

That brings us back to explaining how the PIDL returned by SHGetSpecialFolderLocation() is converted to a pathname. Whenever you're working with different names for the same thing, you need a conversion function. SHGetPathFromIDList() is such a function. If you feed this function with a PIDL, it returns the path and filename of the object called in the PIDL as a C string. Naturally, this is provided that the item identifier list specified by the PIDL is pointing to an object from the file system. This is usually the case with the CSIDL constants called in the previous section. You get the hard drive directory of the object.

```

BOOL SHGetPathFromIDList(
    LPCITEMIDLIST pidl,           // Pointer to item identifier list of object
    LPSTR pszPath                // Pointer to buffer for path and filename
);

```

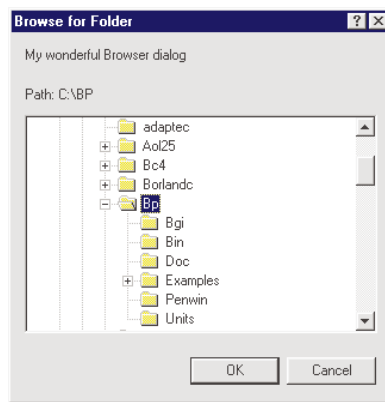
At the very least, you will recognize whether the Shell object really has a match on the hard drive by the function result. The result, in case of error, will be FALSE. If you get a function result of TRUE, you will find the appropriate path in the buffer whose address you specified in the call to pszPath. It should have room for MAX_PATH characters:

Selecting a folder

Use the SHBrowseForFolder() function to give the user the opportunity to select an existing folder from within an application from the Shell's Namespace. The function puts an Explorer-like dialog box on the screen. The user can then select a folder (but not a file) in this dialog box. The caller specifies the starting point of the display within the Namespace. This lets you limit the visible section of the Namespace to a specific drive or the Network Neighborhood. The user doesn't even see the Shell Namespace from the starting point. Only the folders below the starting point can be viewed.

```
LPITEMIDLIST SHBrowseForFolder(
    LPBROWSEINFO lpbi // Pointer to BrowseInfo with the function parameters
);
```

The Browser dialog box



A structure of the BrowseInfo type passes the starting point of the display and a few other parameters to the function. This structure builds the dialog box, lets the user select a folder and returns to the caller after completion of input. It returns a PIDL as a function result. An example could be a pointer to the item identifier list of the Shell object selected by the user. If input was terminated by Cancel, you get a function result of NULL.

The buffer for the item identifier list was allocated by the IMalloc interface of the Shell, which must also release it. However, to do this, you don't necessarily have to address the interface directly. You can instead use the OLE API CoTaskMemFree() function. The BrowseInfo structure determines the display of the dialog box and the execution of the function. Here is its structure:

```
typedef struct {
    HWND          hwndOwner;          // Window handle of parent
    LPCITEMIDLIST pidlRoot;            // PIDL to Shell object as starting point
    LPSTR          pszDisplay name;    // Pointer to buffer that gets display
                                          // names
    LPCSTR         lpszTitle;         // Pointer to C string as title of dialog box
    UINT          ulFlags;            // BIF constants mit type of folder to be
                                          // displayed
    BFFCALLBACK    lpfn;              // Pointer to Callback function of caller
    LPARAM         lParam;            // lparam for the Callback fct of caller
    int            *iImage;           // Address of an int that receives the index
                                          // for icon
} BROWSEINFO, *PBROWSEINFO, *LPBROWSEINFO; // Pointer to BrowseInfo
```

The first parameter receives the window handle of the parent. It also helps the system avoid receiving any more input within the window until the dialog box has been closed again. pidlRoot is a PIDL that points to the item identifier list of the folder with which display is to begin. The item identifier must be defined regarding the Desktop, the root of the Namespace. The following lists the three methods of reaching the PIDL:

1. Use SHGetDesktopFolder(), if you want to let the user browse through the whole Desktop.

2. If you want to start with one of the virtual folders, like the Network Neighborhood, use SHGetSpecialFolderLocation() to get the appropriate PIDL.
3. To have the display start elsewhere within the Namespace, call the various methods of the IShellFolder interface to compile the PIDL. While this involves the most work, it makes up for it by offering the greatest possible flexibility.

In the next parameter, pszDisplay name, you pass the address of a buffer in which the function writes the display name of the selected folder as a C string in case of success. The function assumes the buffer contains a minimum of MAX_PATH characters. Therefore, you don't have to specify a smaller buffer. lpszTitle also represents a pointer to a C string. However, this time also with the title of the dialog box, which can be set by the caller.

A combination of different BIF flags is expected in ulFlags, which determine what kind of folder contents the user sees:

BIF constants for SHBrowseForFolder ()	
Constant	Meaning
BIF_RETURNFSANCESTORS	User can only select drives and servers. The OK button is disabled for all other items.
BIF_RETURNONLYFSDIRS	User can only select drives that are part of the file system. The OK button is disabled for all other items.
BIF_DONTGOBELOWDOMAIN	No folders displayed below domain level. All you see is the network landscape/map.
BIF_BROWSEFORCOMPUTER	User can only select one computer. The OK button is disabled for all other items.
BIF_BROWSEFORPRINTER	User can only select printers. The OK button is disabled for all other items.
BIF_STATUSTEXT	Adds a status line to the dialog box that the caller can fill using a callback.

This is the callback function that can be specified in the call to SHBrowseForFolder() in the lpfn parameter. This function gives lets the caller note different user actions within the dialog box. It also lets the caller act on the controls within the dialog box during input. Before showing how this works, we need to talk about the last parameter of the function, iImage.

The caller must load this parameter with the address of an int variable. The function loads this variable with an index value after successful execution. It's the index of the icon for the folder selected by the user. The index refers to the systemwide valid image list. The Shell keeps all the necessary file and folder icons for Desktop display in this list. For more information, read about the image list in the section on the SHGetFileInfo() API function.

By using callback, the caller can react to user input during display of the dialog box. For example, the caller could use it to disable the OK button, to display a special folder or leave a message in the status line of the dialog box. However, to do this, a callback function of the BFFCALLBACK type must be defined first. This is declared in the following code segment:

```
typedef int (CALLBACK* BFFCALLBACK)(          // BFFCALLBACK is a function pointer
    HWND    hwnd,                          // Window handle of Browse dialog box
    UINT    uMsg,                          // Message
    LPARAM  lParam,                        // extra message parameters depending on uMsg
    LPARAM  lpData                         // the lParam field from BrowseInfo :-(
);
```

The dialog box calls the callback function in two situations:

1. Its called during the initialization of the dialog box.
2. Whenever the user selects a different folder.

Use uMSG to display the base of the call: BFFM_INITIALIZED during initialization and BFFM_SELCHANGED after changing folders. lParam displays additional information, depending on the message. lpData represents the BROWSEINFO.lParam parameter as specified in the call to SHBrowseForFolder(). This way a connection is built between

the caller of SHBrowseForFolder() and the callback function. This puts you in the position of maintaining a type of common/shared context without having to resort to global variables.

The IParam parameter is undefined in the call to the callback function with the BFFM_INITIALIZED message. It's different with BFFM_SELCHANGED, though. The parameter represents a PIDL with this message. So, it points to the item identifier list of the newly selected object. The callback function is not allowed to change this list. It can, however, create copies and pass them to Shell functions or methods of IShellFolder.

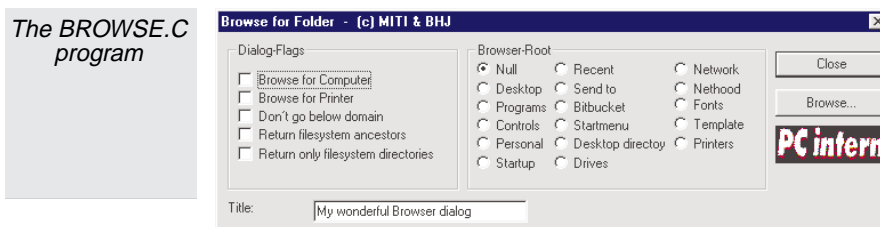
Regardless of the reason you have for calling the callback function, it can always use SendMessage() to send a message to the dialog box. The window handle is passed to the function in the hWnd parameter. Three messages can be sent, offering the caller some interesting options for influencing the dialog box.

Messages to a Browser dialog box		
Constant	Meaning	Background
BFFM_ENABLEOK	Enables the OK button when wParam is set to TRUE, disables the button on FALSE.	Gives an application the option of deciding on its own whether the user can select a specific folder.
BFFM_SETSELECTION	Sets a new folder. To set a path in the file system, specify a pointer to the path in IParam and specify TRUE in wParam. Otherwise specify a PIDL in IParam and FALSE in wParam.	Gives the application the option of setting a different folder during installation and later when the user is selecting a folder.
BFFM_SETSTATUSTEXT	Displays a C string in the status line. IParam must point to a corresponding C string.	Using this message, the application can communicate at any time with the user and make comments on the user's selection.

Using the Browser dialog box

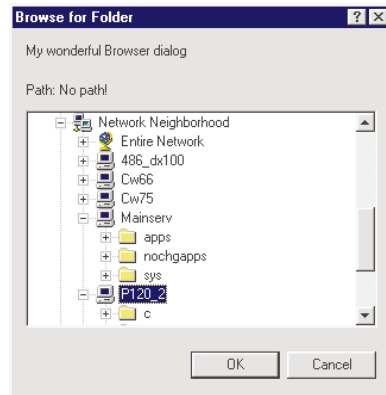
The BROWSE.MAK program shows working with the Browser dialog box and the call to the SHBrowseForFolder() function. The main focus of the program is a file called BROWSE.C.

The program provides an easy method for browsing through various areas of the Shell Namespace using SHBrowseForFolder(). The following illustration shows how the "Browser Root" dialog box specifies several starting points. They all correspond to one of the various CSIDL constants. If the application clicks the [Browse...] button, a PIDL is determined first by calling SHGetSpecialFolderLocation() in conjunction with the selected CSIDL constant. Then SHBrowseForFolder() is called with the returned PIDL so the desired area from the Shell Namespace appears on the screen.



Also, the different BIF flags can be selected in the "Dialog-Flags" dialog box. These BIF flags are passed to SHBrowseForFolder() when it is called in the uFlags field of the BROWSEINFO structure. You can now easily understand the effects these flags have on the display of the various folders.

BROWSE.C was used here to view the part of the Shell Namespace that represents the network



The listing of the BROWSE.C module shows that BROWSEINFO uses the option for defining a callback function. BrowseCallbackProc() is the name of the function used here to display the pathname of the selected folder on disk within the Browser dialog box. If the folder doesn't have a match in the file system, "No path" appears there.

You'll find the following program(s) on the companion CD-ROM



BROWSE.C (C listing)

Copying and other file operations

The Shell API also has a function for developing elementary file operations such as copying, deleting, moving and renaming files. This function saves the caller a great deal of work. This is especially true since the Shell uses this function itself so you can assume that it runs properly. Also, the customary dialog box with the flying files appears on the screen showing the progress of the operation.

```
int SHFileOperation(
    LPSHFILEOPSTRUCT lpFileOp           // Pointer to SHFILEOPSTRUCT
);
```

The execution of the function is determined by the pointer to a SHFILEOPSTRUCT type structure. The caller must initialize it before the function call with the desired operation and the involved files. The following is the structure:

```
typedef struct {
    HWND      hwnd;           // Window handle of parent
    UINT      wFunc;          // Operation to be executed (FO_COPY, FO_MOVE etc.)
    LPCSTR     pFrom;          // Pointer to C-Str with name of source file(s)
    LPCSTR     pTo;           // Pointer to C-Str with name of destination file(s)
    FILEOP_FLAGS fFlags;       // FOF-Flags
    BOOL       fAnyOperationsAborted; // Displays cancel. by user after
                                // call
    LPVOID     hNameMappings;  // Receives handle of name map after call
    LPCSTR     lpszProgressTitle; // C-Str for display (FOF_SIMPLEPROGRESS)
} SHFILEOPSTRUCT, *LPSHFILEOPSTRUCT;
```

A window handle is expected in the first parameter. It must serve as parent for the display of the dialog box. Then one of the FO constants that describes the operation to be executed is expected in wFunc.

pFrom and pTo represent pointers to C strings that contain the names of the source and destination files of the desired operation. Both strings can name several files in sequence. Each file must be listed as an independent string ended by a NUL character. The last filename must be followed by a blank string that consists of a NUL byte. In this way the function is able to detect the end of the list. The current path from the drive is considered to be the starting point if the names do not contain a path. The current drive is used when there is no drive specification either.

FO constants for SHFileOperation()	
Constant	Represents
FO_MOVE	Move files
FO_COPY	Copy files
FO_DELETE	Delete files
FO_RENAME	Rename files

The caller supplies additional information. This information may, for example, show by using the `fFlags` field in `SHFILEOPSTRUCT` how the desired file operation is to be executed. You can specify a combination of the following FOF constants.

FOF constants for SHFileOperation()	
Constant	Represents ...
FOF_SIMPLEPROGRESS	Only a simple dialog box appears as a progress indicator. Unlike a normal progress dialog box, the name of the file currently being edited does not appear in this dialog box. Instead, the string called in <code>SHFILEOPSTRUCT.lpszProgressTitle</code> appears.
FOF_SILENT	During execution, no dialog box will appear as a progress indicator.
FOF_NOCONFIRMATION	"YES to all" is automatically chosen for each access requiring user confirmation.
FOF_NOCONFIRMMKDIR	If a new directory must be created, the user doesn't have to give confirmation.
FOF_MULTIDESTFILES	If several files are called in <code>pFrom</code> , this flag indicates that the first string in <code>pTo</code> is not to be interpreted as the directory to which the files are copied. Instead, for each file in <code>pFrom</code> , <code>pTo</code> contains an entry as destination name of the file.
FOF_RENAMEONCOLLISION	When you attempt to create a file that already exists, an alternative name is used which refers to or indicates the original filename.
FOF_WANTMAPPINGHANDLE	Specify this flag to have the function fill the <code>SHFILEOPSTRUCT.hNameMappings</code> field during the function call. This gives the caller a look at the edited files and what has become of them.

Some of these flags only specify the screen display during file operations. In other words, whether a dialog box will be displayed and, if so, what type of dialog box will be displayed. In default view (without specifying `FOF_SILENT` or `FOF_SIMPLEPROGRESS`) the same dialog box appears that the Shell displays for a manual file operation. Other FOF flags answer the question of whether the user must confirm creating new directories and overwriting files.

`FOF_WANTMAPPINGHANDLE` is a special feature. By specifying this flag, the caller commits the function to account for the executed file operations. For this purpose, the function returns a handle to a name mapping object in `SHFILEOPSTRUCT.hNameMappings`. The object contains an `SHNAMEMAPPING` type structure for each edited file. It lists the old and new path of a file.

```
typedef struct {
    LPSTR    pszOldPath;           // Pointer to old filename
    LPSTR    pszNewPath;          // Pointer to new filename
    int      cchOldPath;           // Character in old filename
    int      cchNewPath;          // Character in new filename
} SHNAMEMAPPING, FAR *LPSHNAMEMAPPING;
```

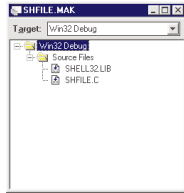
However, you cannot directly access the individual `SHNAMEMAPPING` entries in the returned object. Instead, you must use two macros that are defined in `SHELLAPI.H`: `SHGetNameMappingCount()` and `SHGetNameMappingPtr()`. The first macro returns the number of available `SHNAMEMAPPING` structures. The second macro returns a pointer to an `SHNAMEMAPPING` structure specified by its index. The index must move itself between 0 and the result of `SHGetNameMappingCount() - 1`.

```
#define SHGetNameMappingCount(_hnm)    DSA_GetItemCount(_hnm)
#define SHGetNameMappingPtr(_hnm, _iItem)
    (LPSHNAMEMAPPING)DSA_GetItemPtr(_hnm, _iItem)
```

Also, the handle of the object from `SHFILEOPSTRUCT.hNameMappings` must, in both cases, be specified as the first parameter. This makes it possible to monitor the comings and goings of the various files exactly. If you no longer require the information, you must free the Name-Mapping object again. A function called `SHFreeNameMappings()` is available for this purpose, which expects the value from `SHFILEOPSTRUCT.hNameMappings` as a parameter.


```
void SHFreeNameMappings(
    HANDLE hNameMappings    // Handle of Name-Mapping object to be released
);
```

*Files in the
SHFILE.MAK
project*



Sample program

Naturally, the SHFileOperation() function, together with the new Common Controls, is ideal for reproducing the functionality of the Explorer within an application. Furthermore, the function does excellent work when files must be copied, moved or renamed automatically from a program. For example, you could use this function for installation programs or for automatic backup.

An example is SHFILE.C. It automatically copies all .C, .H and .MAK files from the current directory to C:\PC Intern Backup. SHFileOperation.c automatically creates the directory if it doesn't exist. You'll find this program on the PC INTERN CD in a directory called \WIN95\SHL.

*You'll find the following program(s) on
the companion CD-ROM*



SHFILE.C (C listing)

Getting information about files and Shell objects

With SHGetFileInfo(), the Shell API is providing a function you can use to determine all conceivable attributes about objects from the file system or the Shell Namespace. The function does all of that without OLE objects. For example, you can find out whether an executable file is a DOS, 16 bit or a 32 bit Windows application. Alternatively, you can determine the type of file you have according to the file extension. This function is also the best way to determine an object's icon.

As its first parameter, SHGetFileInfo() expects the name of the object for which you want information. To access a file or directory in the file system, specify a pointer to a C string with path and filename here. Both absolute path specifications and relative path specifications are permissible. Relative paths are seen for the current directory on the drive being addressed. On the other hand, specify a PIDL in pszPath to reference an object from the Shell Namespace. The pointer must point to an item identifier list that describes the desired object relative to the root of the Namespace. So you always use an absolute path specification with PIDLs.

```
DWORD SHGetFileInfo(
    LPCSTR    pszPath,                // Pointer to C string with path or PIDL
    DWORD     dwFileAttributes,      // reserved, 0
    SHFILEINFO *psfi,                // Pointer to SHFILEINFO
    UINT      cbFileInfo,            // Number of bytes in SHFILEINFO
    UINT      uFlags                  // Desired object attributes
);
```

The Attributes

The uFlags parameter determines which type of attribute the function displays. A series of SHGFI constants is available for definition. These constants are organized a bit chaotically because the function is designed to carry out totally different tasks and developers attempted to reconcile everything through the SHGFI constants. Therefore, various commands are also considered to be commands and trigger additional actions. For example, SHGFI_PIDL occupies a special position. You need to add this flag when you specify a PIDL in pszPath instead of a path and filename. Additionally, several flags are available to query an object's icon. Also, two flags you can use to determine the display name and type description of an object. Finally, you can also find out what type of file you have, provided it is an executable file. That way you can tell whether you are dealing with a DOS application, a Win16 or a Win32 application. We'll go into more detail about the different uses in a minute.

First, let's look at the other parameters of SHGetFileInfo(). You don't need dwFileAttribs and can specify 0 for it. psfi and cbFileInfo refer to a SHFILEINFO type data structure that the caller must pass to the function so that it can return information to the caller inside it. That's why you don't need to initialize this structure before the function call.

However, SHGetFileInfo() doesn't fill in all the fields of the structure. Instead it fills only those fields whose contents were requested by the SHGFI flag. The first two parameters return an icon. The first parameter receives the icon handle. The second parameter receives an index to a file or an image list. In the third parameter, dwAttributes is returned when the SHGFI flag, SHGFI_ATTRIBUTES is specified. Also, SFGAO constants (these are the ones from IShellFolder::GetAttributesOf) are returned in the third parameter. These constants describe the attributes of the object. szDisplayName only receives the address of a C string that holds the display name of the object. szTypeName receives the type of object.

```
typedef struct {
    HICON hIcon;                //Handle of returned icon
    int iIcon;                  // Index of returned icon
    DWORD dwAttributes;         // Attributes of the object (SFGAO constants)
    CHAR szDisplayName[260];     // Buffer for display names with MAX_PATH
                                // length
    CHAR szTypeName[80];        // Buffer with 80 bytes for the type name
} SHFILEINFO;
```

The call to SHGetFileInfo() passes the address of the allocated SHFILEINFO structure in psfi, and passes its size in cbFileInfo. By setting this information, the function is attempting to make certain the required space for placing this information really exists. As a rule you initialize cbFileInfo with sizeof(SHFILEINFO).

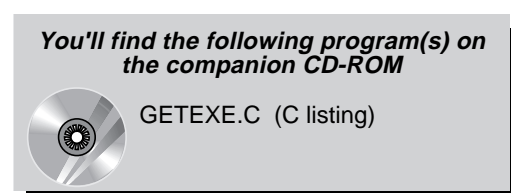
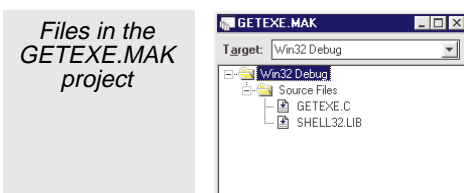
Now let's turn our attention to the various SHGFI constants and the options they offer. The simplest operation is using the two flags SHGFI_DISPLAYNAME or SHGFI_TYPENAME to query the display and type name. You can also combine them to kill two birds with one stone. You'll find the display name after the function call in the SHFILEINFO.szDisplayName field. The type name is in SHFILEINFO.szTypeName.

Querying the EXE type

Next flag: SHGFI_EXETYPE. Use this to determine what type of executable file you have, provided the specified object is indeed an executable. The structure won't tell you whether a file is executable. Instead, the return value provides this information. Thus in this case, you can specify NULL for the psfi pointer. If the specified file is not an executable, you get a function result of 0. Otherwise, you get code/an ID in the lo word and hi word of the returned DWORD. Two ASCII characters characterizing the type of EXE file are in the Lo word. The high word also has additional information.

Using SHGetFileInfo() to interpret the function result from a query for the type of executable file		
Lo-Word	Hi-Word	Meaning
„MZ“	0	DOS application, an EXE, COM or BAT file
„NE“	Hi byte contains the main version number and the Lo byte contains the subversion number of the minimum Windows version required.	16-bit Windows application
„PE“	Hi byte contains the main version number and the Lo byte contains the subversion number of the minimum version of Windows required for execution.	32-bit Windows application
„PE“	0	Win32 console application

The following small program, GETEXE.C, demonstrates the query for an application's EXE type. Since it's a console application, specify the name of the file whose Exe type you wish to determine in the call.



Attributes and Icons

Specify the SHGFI_ATTRIBUTES flag to determine the attributes of the object, as the GetAttributesOf method of the IShellFolder interface returns. After SHGetFileInfo() returns, you will find the attributes of the object in SHFILEINFO.dwAttributes, as IShellFolder::GetAttributesOf returns them. For example, by testing flags like SFGAO_LINK or SFGAO_HASSUBFOLDER you can determine whether the object (or file) is a shortcut or a folder that possesses a subfolder. Also, you can query all the other SFGAO constants through simple AND binary operations.

Finally, we get to the icons. By specifying one of the three flags SHGFI_ICON, SHGFI_ICONLOCATION or SHGFI_SYSICONINDEX you can determine the specified object's icon. The three flags differ only in the kind of information they return about the icon and in which field of the SHFILEINFO structure you find them after the function call:

SHGFI_ICON	You get three items of information by specifying this flag: (1) The icon handle of the object in SHFILEINFO.hIcon; (2) The index of the icon in the system image list in SHFILEINFO.iIcon; (3) The handle of the system image list as the function result of the SHGetFileInfo() call. The system image list is a common control of the image list type in which the Shell keeps all icons displayed on the Desktop. As a result, you can access the desired icon using the returned icon handle. Alternatively you can load it in combination directly from the system image list using the handle of the system image list and the corresponding index.
SHGFI_SYSICONINDEX	Identical to SFGI_ICON, except you don't get the icon handle but only the index of the icon within the system image list in SHFILEINFO.iIcon and the handle of the system image list as a function result.
SHGFI_ICONLOCATION	Determines the path and name of the file in which the icon is located. Returned in SHFILEINFO.szDisplayName.

Along with the three flags, you can specify other flags that determine the type of icon that is returned:

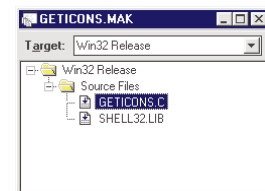
SHGFI_LARGEICON	Returns the 32 * 32 dot default icon.
SHGFI_SMALLICON	Instead of the default icon, the small 16 * 16 dot icon is returned.
SHGFI_LINKOVERLAY	Adds link overlay arrow (icon) to returned icon. The link overlay arrow describes the shortcut.
SHGFI_SELECTED	Returns the icon of the object in selected state.
SHGFI_OPENICON	Returns the icon of the object in opened state (only for folders).

The return value is handled in different ways, depending on the queried information. Normally 0 indicates an error, while all other values represent a successful execution. As we have seen, this is not the case with the specification of SHGFI_EXETYPE and SHGFI_ICON.

GETICONS.C

The companion CD-ROM includes a program for querying the icons. It's called GETICONS.C. It opens one of the default dialog boxes after startup for selecting a file. After the user selects a file, the program displays the file's icons and copies them to the Clipboard. It copies the small icon, the default icon and the default icon with the added link overlay arrow. In the process, the three icons are combined into a bitmap. This lets you insert them easily in any drawing program. For example, you could use Paint, as shown in the following screen shots (including the small illustration, above right). You will find the program on the PC INTERN CD in the \WIN95\GETICONS directory.

*Files in the
GETICONS.MAK
project*

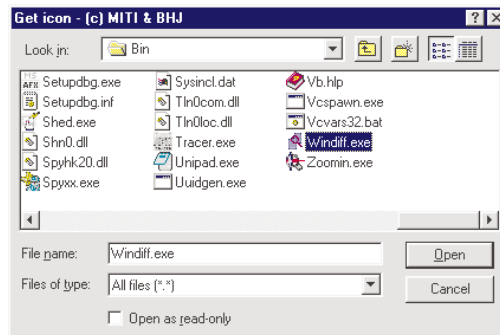


**You'll find the following program(s) on
the companion CD-ROM**

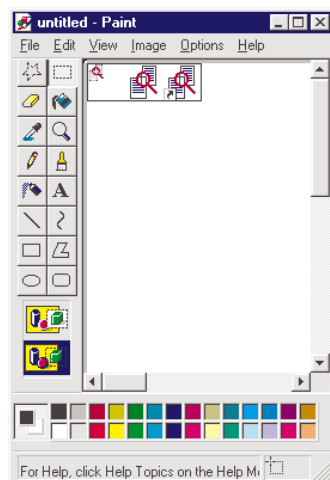


GETICONS.C (C listing)

A file is selected first with Within GetIcon (the file is WINOFF included in Microsoft Visual C++ 2.0)



After the selection is completed, GetIcon copies the three icons to the Clipboard and terminates. Use Paint to paste the icons from the Clipboard



You can also choose a group of icons in the GETICON file selection box. The program will combine their icons into a bitmap. This way you can get a dozen icons in one pass.

The program first opens a default dialog box for file selection through `GetOpenFileName()`. The selected files (or their names) are then passed to the `CopyIconsToClipboard()` function. By using the `CountFileNames()` function, the number of files is determined first. Then a memory bitmap is generated. This bitmap is sized so it can hold three icons for each file. These icons can have a total width of 96 dots and a height of 32 dots. The bitmap is filled with a white brush and then the program browses through the individual files.

For each file, the three icons are determined using `SHGetFileInfo()`. The icon is pasted into the generated bitmap through the returned icon handle and the GDI `DrawIconEx()` function. This bitmap moves to the Clipboard at the end of the function but before the program terminates.

Reading an icon from a file

`SHGetFileInfo()` is an easy way of getting the icon of a file or an object in the Shell Namespace. However, if you wish to access the icons of an EXE or DLL file, use a function called `ExtractIcon()` instead.

```
HICON ExtractIcon(
    HINSTANCE hInst,           // Instance handle of calling process
    LPCSTR lpszExeFileName,    // Pointer to C string with file name
                                // and path
    UINT nIconIndex            // number of icon to be extracted
);
```

Specify the name of an EXE or DLL file as well as the name of icon files as the name of the file. Specify the number of the desired icon within the file in `nIcon` index. Specify 0 for the first, 1 for the second, etc. If successful, you get the handle of the desired icon as the function result. Otherwise, in case of error, you get `NULL`. For example, the system uses this option in conjunction with the `MORICONS.DLL` file. It consists of only one resource with the many icons contained within. Through this icon `ExtractIcon()` loads the icons that the system requires.

Finding the executable file for a document

If you have a filename, for example a text document's filename and you want to determine its "server" application, use the FindExecutable() function. This function searches for the server application by browsing through the registry for a server entry with the appropriate extension of the specified filename. If it finds the extension, you get the name and path for the file in the buffer to which lpResult is pointing.

```
HINSTANCE FindExecutable(
    LPCSTR  lpFile,           // Pointer to C string with file whose server you
                             // are looking for
    LPCSTR  lpReserved,       // NULL
    LPSTR   lpResult          // Pointer to buffer, in which program name is to
                             // be placed
);
```

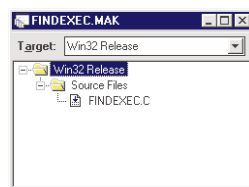
As its first parameter, the function must include the address of a C string which the caller loaded with the path and filename of the file for which the server is needed. If there is no pathname, the program searches the current directory for the file. The third parameter, lpResult, must point to a buffer that the caller allocated. The buffer also must have a minimum of MAX_PATH characters worth of space. If successful, the function places the name and path of the EXE file that can be used to process the specified document. You can tell whether the function was successful by the function result. The result, contrary to the statement of the prototype, does not represent an instance handle. It instead returns a numeric value as error code. Any value less than or equal to 32 represents an error. All other values indicate success.

Pay special attention to error code 31. It indicates the specified file was found. However, no server is listed in the registry for its file extension.

Searching for the Server

The FINDEXEC.C program demonstrates the use of the FindExecutable() function. This program is a console application that expects the name of the file for whose server application you are searching. If able to find one, the program outputs its filename including its path on the console.

*Files in the
FINDEXEC.MAK
project*



**You'll find the following program(s) on
the companion CD-ROM**



FINDEXEC.C (C listing)

Running programs and shortcuts

To start a program or run a shortcut, use the ShellExecute() function. You can also use this function to print documents.

```
HINSTANCE ShellExecute(
    HWND      hwnd,           // Window handle of parent
    LPCSTR    lpOperation,    // Pointer to C string with operation to be
                             // executed
    LPCSTR    lpFile,         // Pointer to C string with filename
    LPSTR     lpParameters,   // Pointer to C string with command line arguments
    LPCSTR    lpDirectory,    // Pointer to C string with working directory
    INT       nShowCmd        // one of the SW-constants for the window display
);
```

ShellExecute() first expects the window handle of the caller. Then it can reference the window handle if it is necessary to display some dialog boxes when executing the function. It then expects a pointer to a C string specifying the operation to be executed. Specify “open” if you wish to start the program, or “print” if you wish to use the printer. Instead of “open”, you can also specify NULL for lpOperation. Either way provides the same result.

lpFile points to the C string that specifies the path and name of the file. If the file is not an executable file, the program looks for the server of the document type determined by the file extension. The server is started with specification of the filename, so that it loads the desired file. If it is a shortcut, the file to which the shortcut points is opened or printed.

To specify command line parameters, place them in a C string and pass their address in lpParameters. You can also define the working directory of the application using lpDirectory. The value specified for nShowCmd determines how the application window appears. You can specify one of the SW constants here, such as SW_SHOWMINIMIZED, SW_SHOWNORMAL etc. For more information, see the description of CreateProcess(). If it’s possible to start the desired program, you get the process handle of the application as a function result. The value of the handle must be greater than 32. All smaller values point to an error in execution. This indicates the desired application could not be started.

Updating the Namespace

The functions of the Shell API and the methods of the Shell interface allow the caller to change the basic elements of the Shell Namespace. Usually it’s a matter of operations performed on files, directories and drives. For example, you might create a new directory in the hard drive directory that represents the program groups under the Start/Programs menu.

The Shell doesn’t remain unaware of these changes for long, however. It might take until the next access to the object before the Shell reads the object again, thus registering the changes. To prevent users from thinking they have started an operation that apparently doesn’t occur in the Desktop, a program can help the Shell somewhat with SHChangeNotify(). SHChangeNotify() forces the Shell to rebuild specific parts of the Namespace, automatically adapting them to the current circumstances.

```
void SHChangeNotify(
    LONG        wEventId,           // what happened? (SHCNE constants)
    UINT        uFlags,             // Meaning of dwItem1 and dwItem2
    LPCVOID     dwItem1,           // Meaning varies, depending on uFlags
    LPCVOID     dwItem2
);
```

The Shell determines what happened through wEventID, allowing it to limit the updating to specific areas of the Namespace. A long list of SHCNE constants is specified, which describe the various events. You will never meet many of these events, such as the docking and removal of network drives.

However, the Shell isn’t satisfied with the information that a change has taken place. It also wants information about what has changed or where changes have been made. Therefore, you have the option of using dwItem1 and dwItem2 to give a more exact description of the changed object. In practice, you will only need dwItem1 to pass either a pointer to a C string with a drive, path and possibly a file specification, or a PIDL. The PIDL must point to an item identifier list that describes the object in question relative to the root of the Namespace.

Use the uFlags parameter to inform the function about which option you choose. There are two constants available for this purpose: SHCNF_IDLIST (PIDL) and SHCNF_PATH (Pointer to C string). Unfortunately, you won’t be able to determine whether the Shell can do anything with this information. The function doesn’t supply a return value.

Constants for SHChangeNotify().	
Constant	Meaning
SHCNE_RENAMEITEM	An item within a folder has been renamed
SHCNE_CREATE	A file has been created.
SHCNE_DELETE	A file has been deleted.
SHCNE_MKDIR	A new directory has been created.
SHCNE_RMDIR	A directory has been deleted.
SHCNE_MEDIAINSERTED	Added removable medium
SHCNE_MEDIAREMOVED	Removed removable medium
SHCNE_DRIVEREMOVED	Network drive removed
SHCNE_DRIVEADD	Network drive added
SHCNE_NETSHARE	Network resource released for sharing
SHCNE_NETUNSHARE	Network resource excluded from sharing
SHCNE_ATTRIBUTES	File attributes have been changed
SHCNE_UPDATEDIR	Contents of a directory have been changed
SHCNE_SERVERDISCONNE	Network server has been disconnected
SHCNE_UPDATEIMAGE	Icon in system image list has been changed
SHCNE_RENAMEFOLDER	Folder has been renamed

SHAddToRecentDocs

It's very easy to add new entries to the group of most recently addressed documents. This is shown in the Start menu under Documents. All you need for this is either the name and path of a file or a PIDL to an item identifier list. This must describe the object within the Shell Namespace, relative to the Desktop object at the root of the Namespace.

```
void SHAddToRecentDocs(
    UINT    uFlags,    // SHARD_PATH for file or SHARD_PIDL for Shell object
    LPCVOID pv // Pointer to C string with path or filename or item ID list
);
```

Interestingly enough, this time no return value is returned. So, you don't know whether the function call was successful. Those who don't like error checking anyway won't be upset about this (just don't tell your boss).

Use `uFlags` in the function call to indicate whether the entry (it's a shortcut to the object) is to be created from a path and filename or from a PIDL. `SHARD_PATH` represents a file and `SHARD_PIDL` represents the PIDL. The pointer must be passed in `pv`.

ShellFolder Objects And The IShellFolder Interface

The PIDL is required before you can edit items of the Desktop from a program. You cannot give an object a unique name without the PIDL. Therefore, you cannot give the object any instructions. Some functions from the Shell API along with the `ShellFolder` object with its `IShellFolder` interface serve as an interface between an application and the Shell Namespace. Together, they offer an application the greatest functionality that is available to users in the structure of the Desktop. All the items within the Shell Namespace can be browsed, observed, and to a certain extent, even changed. For example, you can change names, create new items from a program, such as folders, shortcuts or program groups and their entries. It's also possible to select the current printer, call an application from the Control Panel or display a Browser dialog box with a default starting point. In other words, with this functionality you can do everything that you normally do by hand.

Methods of IShellFolder	
IShellFolder	Task
EnumObjects	Browses the contents of a folder.
BindToObject	Opens a folder.
GetAttributesOf	Gets the attributes and the type of a ShellFolder object.
GetDisplayNameOf	Supplies the display name of a ShellFolder object.
SetNameOf	Sets the display name of a ShellFolder object.

Accessing the objects in the Namespace

To access an object in the Shell's Namespace, start at the root of the Namespace. Work your way to the object, step by step. First, you'll need a ShellFolder object; without it you can't even get to the IShellFolder interface. Therefore, as a rule you start by calling the API SHGetDesktopFolder() function. This function supplies a ShellFolder object, which represents the Desktop as the root of the entire Namespace. Then you can browse through the folder and file objects within the Desktop using the methods of the IShellFolder interface. To continue searching within a specific folder, create a new ShellFolder object through this folder. Then start browsing through the objects. In this way you can work your way to an object level-by-level. Here's the SHGetDesktopFolder() function.

```
HRESULT SHGetDesktopFolder(           // supplies Desktop ShellFolder object
    LPSHELLFOLDER *ppshf // Variable that receives pointer to IShellFolder
);
```

If the call is successful, make certain to release the returned interface pointer before completing your work. To do this, call the Release method.

Browsing through the contents of the folder

The most elementary function that you can execute with this object is the call to the EnumObjects method. It helps you browse through the contents of a ShellFolder object.

```
HRESULT EnumObjects( // generates enumerator for access to folder contents
    IShellFolder *This,
    HWND         hwndOwner, // window handle of parent
    DWORD        grfFlags,   // SHCONTF... constants
    LPENUMIDLIST *ppenumIDList // address of variable that receives the
                                // interface pointer
);
```

Like most methods from the IShellFolder interface, EnumObjects expects a window handle after the obligatory this pointer. The window handle is required in case it becomes necessary to display a dialog box during a Desktop operation. For example, you might need a dialog box to type in a password for access to a network drive or because the user must be prompted to insert a diskette.

Constants for setting folder contents to be browsed	
Constant	Items to browse
SHCONTF_FOLDERS	All folders
SHCONTF_NONFOLDER	All nonfolders, i.e., files and other file objects
SHCONTF_INCLUDEHIDDEN	Include hidden items and system objects

The grfFlags parameter with the SHCONTF constants determines which items within the folder are to be browsed. Any combination of these SHCONTF constants is possible. The last parameter must represent the address of a variable that is to receive a pointer to an IEnumIDList interface. The EnumObjects method makes the returned objects available through an OLE object of the EnumIDList type. This method returns a pointer to the IEnumIDList interface of the generated object to the caller in the specified variable. This interface lets you determine the returned objects.

The IEnumIDList Interface

The IEnumIDList interface follows the structure of a series of enumerator interfaces which OLE defines for browsing through other object sets. This interface contains the elementary methods that assist users in acting on a predefined set of objects (see table on the right).

Methods of IEnumIDList	
IEnumIDList	Task
Clone	Creates a copy of the object.
Next	Inputs the next n entries
Reset	Returns to the beginning of enumeration
Skip	Skips n entries of enumeration

Before the end of the program you must release the interface using its Release method. This also applies to the interface that you get using the Clone method. Clone creates a copy of the object with the objects it contains. Then it returns a pointer to the IEnumIDList interface of the new object. The new object's contents are independent of the old object's contents. It makes sense to use this method whenever you are browsing a path within the Shell Namespace and want to note specific areas to return to at a later time.

```
HRESULT Clone(                                // generates an identical but independent copy
    IEnumIDList *This,
    IEnumIDList **ppenum                     // pointer to variable for receiving new
                                           // interface pointer
) ;
```

Use the Next method to display the entries within the enumeration. You can have one or more entries returned per call. The method always starts after the entry where it stopped during the last call. So you can display the various entries in sequence by making consecutive calls to IEnumIDList::Next. To start over at the beginning, simply call the Reset method. Then, the next time you call the Next method, you will get the first entry again.

```
HRESULT Next(                                // returns the next n entries of the enumeration
    IEnumIDList *This,
    ULONG celt,                             // number of desired entries
    LPITEMIDLIST *rgelt,                    // pointer to array that is to receive PIDs
    ULONG *pceltFetched                      // pointer to Var for number of loaded
                                           // entries
) ;
```

Along with the number of desired entries, Next must be passed a pointer to an array in which the method places a PIDL for each entry. The PIDL specifies the path of the object relative to the parent object. It doesn't contain the complete path of the root of the Namespace. Instead it only contains the item identifier that characterizes the object within its parent folder. The ShellFolder object is described as the parent folder. You called the EnumObjects method from this folder and received the IEnumIDList interface, whose Next method you are now calling. This distinction is important because you may want to note the entire path of an object while working your way from level to level. To do that, you must put together the item identifiers that you got at the different levels through Next. We'll introduce you to the appropriate source code later in the chapter.

You also have to specify the address of a ULONG variable so that you know how many entries were returned. The function enters the number of returned entries. Therefore, the number of array entries are filled with PIDLs in this variable. It doesn't always have to match the requested number. This is especially true where you request more than one entry but you're already too close to the end of the enumeration. If you find 0 in this variable after that, it means you have already reached the end of enumeration.

You can start all over at the beginning with Reset. As we mentioned, Reset sets back the internal counter of the IEnumIDList interface so the next Next call returns the first entry of the enumeration.

```
HRESULT Reset(                                // has Next start over with the first entry
    IEnumIDList *This
) ;
```


Finally, there's the Skip method. By using this method you can skip any number of entries you please. The next Next call will then return the first object after the last skipped object. This is practical when you want to pick out a specific entry of the enumeration. First, call Reset to make sure you are at the beginning of the enumeration. Then call Skip with the number of the entry that you want to retrieve. Finally, call Next to display the entry.

```
HRESULT Skip(                                     // skips N entries of enumeration
    IEnumIDList *This,
    ULONG celt                                     // number of entries to be skipped
) ;
```

Let's turn back to IShellFolder. IShellFolder::EnumObjects and the EnumIDList interface that results from it let you browse through the contents of a ShellFolder object. The only question is, what to do with these objects if you can't manipulate them yourself and you must treat them as a black box. The answer's not difficult from an object oriented standpoint. Use the corresponding methods from the IShellFolder interface on them. Two operations are noteworthy. The first is the output of display names. The second operation is the opening of the object to penetrate one level further within the hierarchy of the Namespace.

Determining the display names

You can get the display name of an item identifier by using the GetDisplayNameOf method of the IShellFolder interface. Some examples of display names are "Desktop", "Start", "(c:)" and "Recycle Bin". To do this, pass a PIDL that points to the item identifier list with the object to the GetDisplayNameOf method. The object must be defined in it relative to the folder object through whose interface you call the GetDisplayNameOf method. If the interface comes from the Desktop object that you found through SHGetDesktopFolder(), you can specify an absolute path. However, if the interface comes from the parent object of the object being addressed here, the item identifier list can only contain an item identifier that describes the object relative to its parent (plus the terminating NUL item identifier). As a result, it's very easy to use this method on the objects that you received in the context of IShellFolder::EnumObjects.

```
HRESULT GetDisplayNameOf( // returns the display name of the folder or file
    IShellFolder *This,
    LPCITEMIDLIST pidl,           // pointer to item ID list of object
    DWORD uFlags,                 // combination of SHGDN constants
    LPSTRRET lpName // pointer to STRRET Var, which gets display name
) ;
```

Specify the type of returned display name using the uFlags parameter. The following SHGDN constants are available.

Constants for IShellFolder::GetDisplayNameOf	
Constant	Meaning
SHGDN_NORMAL	The normal display name, as it appears on the Desktop.
SHGDN_INFOLDER	The display name relative to the parent folder. It's usually identical to the "normal" display name.
Reset	Returns to the beginning of enumeration

Finally, use the last parameter to specify the address of a STRRET type variable that is to be loaded by the GetDisplayNameOf method with the display name of the object. STRRET is not just another type for a string pointer. Instead, it's a data structure for managing strings in different formats. You can determine the format of the returned display name from the uType field. It represents one of the three constants STRRET_WSTR, STRRET_OFFSET and STRRET_CSTR.

The following shows how it's defined in SHLOBJ.H.

```
// Constants for STRRET.uType —————

#define STRRET_WSTR    0x0000    // UNICODE string (STRRET.pOleStr)
#define STRRET_OFFSET 0x0001    // ANSI C string (STRRET.uOffset)
```

```

#define STRRET_CSTR    0x0002        // STRRET is cStr is ANSI C string

// the STRRET type -----
typedef struct _STRRET
{
    UINT    uType;                    // one of the STRRET_ constants
    union
    {
        LPWSTR  pOleStr;              // UNICODE string
        UINT    uOffset;              // Offset in PIDL
        char    cStr[MAX_PATH];       // here's the string itself
    } DUMMYUNIONNAME;
} STRRET, *LPSTRRET;

```

The easiest scenario is when `uType` contains the value of `STRRET_CSTR`. Then the `cStr` field will have the display name in ANSI format as a C string. It's a bit more difficult with `STRRET_OFFSET`. In this case, the `uOffset` field specifies the distance (the offset) of the string relative to the PIDL that you specified in the call to `IShellFolder::GetDisplayNameOf`. `STRRET_OFFSET` is normally used when the desired display name is already a C string in ANSI format within the item identifier. Here's what it looks like:

```
char * DisplayName = ((char * ) PIDL + strtret.uOffset)
```

The most time-consuming of all is when `uType` displays `STRRET_WSTR`. That means you are dealing with a Unicode string. It must first be converted to a normal ANSI C string using the `API WideCharToMultiByte()` function. The following function removes much of the frustration of working with `STRRET`. It returns an ANSI string whose memory was dynamically allocated. Later you'll have to release the memory again through `CoTaskMemFree()`. We took the function from the `EnumFolder.c` help folder. Several programs in this chapter are based on this folder.

```

/*****
/* ExtractStrRet : Gets ANSI string contained in a
/*
/*          STRRET
/*-----*/
/* Parameters:  pIDL      - Address of PIDL belonging to
/*
/*          STRRET string
/*
/*          lpStr      - Address of STRRET string
/*
/* Return value : Address of ANSI string or NULL, in case of error
/*-----*/
/* Info : The caller must deallocate the returned memory
/*
/*          again via CoTaskMemFree().
*****/
LPSTR ExtractStrRet( LPITEMIDLIST pIDL,
                    LPSTRRET lpStr )
{
    LPSTR lpsz = NULL;
    int cch;

    switch( lpStr->uType )
    {
        case STRRET_WSTR:
            // Wide Characters
            // Get number of characters -----
            cch = WideCharToMultiByte( CP_OEMCP,
                                      0,

```

```

        lpStr->pOleStr,
        -1,
        NULL,
        0,
        NULL,
        NULL ) + 1;

    // Allocate memory -----
    lpsz = CoTaskMemAlloc( cch * 2 ); // WORD character
    if( lpsz ) // Copy wide characters to lpsz -----
        WideCharToMultiByte( CP_OEMCP,
                               0,
                               lpStr->pOleStr,
                               -1,
                               lpsz,
                               cch,
                               NULL,
                               NULL );

    break;
case STRRET_OFFSET: // Text contained in PIDL
    // lpStr contains the beginning of first character within the PIDL
    cch = lstrlen( ( ( char* ) pIDL ) + lpStr->uOffset ) + 1;
    // Allocate memory -----
    lpsz = CoTaskMemAlloc( cch );
    if( lpsz )
        strcpy( lpsz, ((char*)pIDL) + lpStr->uOffset); // Copy text
    break;
case STRRET_CSTR: // ordinary C string (how boring!)
    cch = lstrlen( lpStr->cStr ) + 1; // Get length
    // Allocate memory -----
    lpsz = CoTaskMemAlloc( cch );
    if( lpsz )
        strcpy( lpsz, lpStr->cStr ); // Copy text
    break;
}
return lpsz; // Address of ANSI string
}

```

By analyzing the returned display name using such a function, you can easily get the display name of an object within a folder.

Into the depths

The second important operation that you perform on a PIDL is `IShellFolder::BindToObject`. Provided the PIDL is for a folder, you can use it to get to a new `ShellFolder` object that represents the desired folder. In turn, all the methods of the `IShellFolder` interface can be used on that folder. So, you can browse through the objects in the folder, determine their display names or go one level deeper again using `BindToObject`.

```

HRESULT BindToObject( // furnishes interface for accessing object
    IShellFolder * This,
    LPCITEMIDLIST pidl, // pointer to item ID list describing the object
    LPBC pbcReserved, // reserved, NULL
    const IID *const riid, // pointer to interface ID
    LPVOID *ppvOut // pointer to Var that receives the interface

```

```

        // pointer
    ) ;

```

The method expects a pointer in the `pidl` parameter to the item identifier list. It describes the `ShellFolder` object that is to be opened. The item identifier can only consist of one entry that describes the object relative to its parent folder. The following item identifier must end the list by specifying a length of 0.

The address of a variable is expected in `riid` with the interface ID of the interface used to access the new object. Because the object is of the `ShellFolder` type, you will usually want access to the object's `IShellFolder` interface. In this case, you specify the address of the externally declared variable `IID_IShellFolder` for `riid`. It contains the necessary interface ID. The last function parameter, `ppvout`, determines where the pointer to the interface is placed. It must point to a corresponding variable. You can access this variable through the `IShellFolder` interface of the newly opened folder after the successful function call. Because a new object has been created, remember to call the object's `Release` method when you no longer need it. Naturally, you can also delete the parent object from which you called `BindToObject`. Keep in mind that the new object is no longer dependent on its parent object in any way.

The `DD.C` program demonstrates this procedure with concrete code. We'll introduce you to this program at the end of the following section. The program performs a type of `DIR` command for the Desktop. This helps you browse the Desktop console level as you would browse the contents of a hard drive directory using `DIR`.

Finding out an object's attributes

Naturally, when you're browsing the contents of a folder using the `IEnumIDList` interface returned by `EnumObjects`, another folder object may be present. At some point you're bound to run into files, programs and shortcuts. So, before hurrying to call `BindToObject` for an object that doesn't even represent a folder (and therefore cannot be opened in this way), first determine what kind of object it is.

Use the `GetAttributesOf` method to do this. This method is able to determine the attributes of any number of item identifiers at the same time. For this purpose, the method expects a pointer to an array consisting of `PIDL`s so every array entry points to an item identifier list, and so refers to an object. As with the previous methods, here again the desired objects must be defined relative to their parent object. So, the item identifier list of the object can only consist of the object's item identifier and an empty item identifier that follows. In the `cidl` parameter, pass the method the number of objects to be queried and, therefore, the number of array items that will be allocated.

```

HRESULT GetAttributesOf(           // returns attributes of folders or files
    IShellFolder *This,
    UINT cidl,                     // number of pointers in the array
    LPCITEMIDLIST *apidl,         // pointer to array with pointers to item IDs
    ULONG *rgfInOut              // pointer to array with desired attributes
) ;

```

Thus, `GetAttributesOf` turns out to be the ideal partner for `EnumObjects` and the `IEnumIDList` interface which it returns. This interface's `Next` method returns a desired number of objects through an array, which you can then pass on to `IShellFolder::GetAttributesOf` immediately.

However, you will also need an array, whose address is expected in `rgfInOut`. The caller must create this array, which needs to have a corresponding entry for each entry in the `PIDL` array. To reference only one object, simply specify the address of a `ULONG` variable.

Before calling `GetAttributesOf`, initialize this variable or the individual array items from `rgfInOut` with the attributes that you want to get from the object in question. For example, if you have ten objects and want to determine the same items about all ten, place the same attribute values in all ten corresponding `rgfInOut` entries. However, you can also vary the values.

The following table lists the available attributes and constants that represent them. Depending on the attribute for which you want to query, combine the different constants when initializing the array entries through an `OR` operator. As a result, the

method determines the attributes on which it needs to concentrate. As it receives this information, it also returns it. After the call, each item in `rgfInOut` is first set to 0, thus clearing all flags. Then the requested attributes are checked. The constant is inserted again by an OR operator (provided the attribute applies). Each set attribute before the call represents a question, while after the question it represents an answer. Before the call you add an OR, afterwards you check with AND to see whether the flag is set. If the flag is set, the attribute applies.

Considering the number of attributes, this method probably answers many questions about an object in the Shell Namespace. For example, it helps you answer questions like the following:

- Whether the object is a folder, a normal file or a shortcut
- Whether the object can be copied, renamed or moved
- Whether the object physically exists within the file system or was virtually constructed by the Desktop

Constant	Meaning
SFGAO_CANCOPY	Object can be copied.
SFGAO_CANMOVE	Object can be moved.
SFGAO_CANLINK	Shortcuts to the object can be created.
SFGAO_CANRENAME	Object can be renamed.
SFGAO_CANDELETE	Object can be deleted.
SFGAO_HASPROPSHEET	Object has a property sheet.
SFGAO_DROPTARGET	Other objects can be dropped on it.
SFGAO_CAPABILITYMASK	Combines all the flags mentioned above.
SFGAO_LINK	Object is a shortcut.
SFGAO_SHARE	Object is shared on network by several computers.
SFGAO_READONLY	Object is read only.
SFGAO_GHOSTED	Object is displayed by a ghosted icon because it's unavailable and so can't be selected.
SFGAO_DISPLAYATTRMASK	Combines <code>_LINK</code> , <code>_SHARE</code> , <code>_READONLY</code> and <code>_GHOSTED</code> .
SFGAO_FILESYSANCESTOR	Object represents the root of a file system (A:, B:, C:, etc., and UNC names for server).
SFGAO_FOLDER	Object is a folder.
SFGAO_FILESYSTEM	Object represents a physical item of the file system, a file or a directory on a drive.
SFGAO_HASSUBFOLDER	Object has subfolders.
SFGAO_CONTENTSMASK	Combines <code>_FILESYSANCESTOR</code> , <code>_FOLDER</code> and <code>_FILESYSTEM</code> (see comment).
SFGAO_VALIDATE	Validate information already gathered about object.
SFGAO_REMOVABLE	Object on a removable storage medium?

Naturally, the trick with the AND and OR only works because each of the constants is mapped to one of the 32 bit positions of a ULONG so they can't overwrite one another. The three constants `SFGAO_CAPABILITYMASK`, `SFGAO_DISPLAYATTRMASK` and `SFGAO_CONTENTSMASK` saves some typing because they combine the various attributes from one of three groups. Microsoft seems to have made a little mistake with `SFGAO_CONTENTSMASK`, at least in our version of `SHLOBJ.H`. `SFGAO_CONTENTSMASK`. It combines the four constants `SFGAO_FILESYSANCESTOR`, `SFGAO_FOLDER`, `SFGAO_FILESYSTEM` and `SFGAO_HASSUBFOLDER`. However, it should be defined by `0xF0000000L` and not `0x80000000L` as happened here.

```
// Excerpt from SHLOBJ.H

#define SFGAO_FILESYSANCESTOR    0x10000000L
```

```
#define SFGAO_FOLDER          0x20000000L
#define SFGAO_FILESYSTEM     0x40000000L
#define SFGAO_HASSUBFOLDER   0x80000000L
#define SFGAO_CONTENTSMASK   0x80000000L // supposed to be 0xF0000000L
```

Among the various flags, SFGAO_VALIDATE has a special role. It doesn't query for any attribute. It instead instructs the ShellFolder object not to resort to its internal cache in querying for the desired attributes. This is always a good idea when you must assume that the attributes have changed since a prior access to the object. Therefore, this flag instructs the ShellFolder object to get the desired attributes directly from the object.

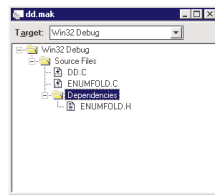
One last flag that we don't want to leave out is SFGAO_FILESYSTEM. In their first programming efforts with the Desktop, users are usually unsure if they're dealing with real objects or virtual objects created by the Desktop. SFGAO_FILESYSTEM usually clarifies this for most users. It displays whether an object is a real item within the file system, i.e., a directory or a file.

DD - a DIR command for the Desktop

The program called DD.C lets you view the contents of the Desktop and its folders from the screen. The program is a type of DIR command for the Namespace. It browses folders and their subfolders, displaying them on the screen with display names and storage location on the hard drive. You can also query the attributes of the items, for example CANCOPY, LINK or HASSUBFOLDER. This gives you more information about the character of the individual items. For example, it helps you tell shortcut files from folders.

As the starting point for display, use all folders represented by one of the CSIDL constants for the call to SHGetSpecialFolderLocation(). However, DD receives the desired starting point without the preceding "CSIDL_". You don't have to worry about upper or lower case letters, either. So CSIDL_DESKTOP becomes desktop and CSIDL_FONTS is simply fonts.

*Files in the
DD.MAK project*



You can determine through the command line whether to browse only the contents of the specified folder or to include its subfolders. Specify the "r" switch to have the program browse all subfolders in a recursion.

The program outputs the browsed folder and file objects in a line on the screen. First the display name appears, then the storage location of the item on the hard drive in brackets. If the object doesn't have a match in the file system, the brackets are empty.

To have a look at the attributes of the folders and their items, specify one or more aAttribut switches. For example, you could specify -acancopy -ahassubfolder. The SFGAO constants are used as attribute names. These constants are like the CSIDL constants. You can omit the prefix and not worry about upper or lower case letters.

The desired attributes are specified under the line with the display name. Only those attributes that are set will appear. Use the -d switch to specify whether to display all browsed folders or only those with the desired attributes. The switch has the following options:

The -d switch for the DD program	
Switch	Meaning
-dx	Display all browsed folder and file objects, regardless of whether they have one of the desired attributes. This is the default setting.
-d-	Display only file and folder objects with at least one of the requested attributes.
-d+	Display only folders and file objects with all the requested attributes.

However, you don't have to memorize the syntax of DD. If you call DD without specifying any parameters, the program will show you its syntax on the screen:

Syntax of DD.C

```

DesktopDir - (c) MITI & BHJ
Call: dd START [-r] [-dTYPE] [-aATTRIB] [-aATTRIB]
-r : Recursion through subfolders
-d : Display-Type
    TYPE : - : Display when at least one attribute set
           + : Display when all attributes set
           x : Always display
START: DESKTOP      ATTRIB: CANCOPY
        PROGRAMS    CANMOVE
        CONTROLS    CANLINK
        PRINTERS     CANRENAME
        STARTUP      CANDELETE
        RECENT       HASPROPSHEET
        SENDTO       DROPTARGET
        BITBUCKET    LINK
        STARTMENU    SHARE
        DESKTOPDIRECTORY READONLY
        DRIVES       GHOSTED
        NETWORK      FILESSANCESTOR
        METHOD        FOLDER
        FONTS        FILESYSTEM
        TEMPLATES    HASSUBFOLDER
                   REMOVABLE

Press any key to continue

```

Calling dd fonts
-r

```

DesktopDir - (c) MITI & BHJ
C:\Win95\Fonts C:\WIN95\FONTS\
Vga850 [C:\WIN95\FONTS\VGA850.FON]
Desktop [C:\WIN95\FONTS\DESKTOP.INI]
Arial [C:\WIN95\FONTS\ARIAL.TTF]
Arialbd [C:\WIN95\FONTS\ARIALBD.TTF]
Arialbi [C:\WIN95\FONTS\ARIALBI.TTF]
Ariali [C:\WIN95\FONTS\ARIALI.TTF]
Cour [C:\WIN95\FONTS\COUR.TTF]
Courbd [C:\WIN95\FONTS\COURBD.TTF]
Courbi [C:\WIN95\FONTS\COURBI.TTF]
Couri [C:\WIN95\FONTS\COURI.TTF]
Times [C:\WIN95\FONTS\TIMES.TTF]
Timesbd [C:\WIN95\FONTS\TIMESBD.TTF]
Timesbi [C:\WIN95\FONTS\TIMESBI.TTF]
Timesi [C:\WIN95\FONTS\TIMESI.TTF]
Wingding [C:\WIN95\FONTS\WINGDING.TTF]
8514fix [C:\WIN95\FONTS\8514FIX.FON]
8514oem [C:\WIN95\FONTS\8514OEM.FON]
8514sys [C:\WIN95\FONTS\8514SYS.FON]
App850 [C:\WIN95\FONTS\APP850.FON]
Cours [C:\WIN95\FONTS\COURS.FON]
Coursf [C:\WIN95\FONTS\COURS.FON]
Modern [C:\WIN95\FONTS\MODERN.FON]
Serif [C:\WIN95\FONTS\SERIF.FON]
Seriff [C:\WIN95\FONTS\SERIFF.FON]
Smalle [C:\WIN95\FONTS\SMALLE.FON]
Smallf [C:\WIN95\FONTS\SMALLF.FON]
Sserif [C:\WIN95\FONTS\SERIFF.FON]
Sseriff [C:\WIN95\FONTS\SERIFF.FON]
Symbol [C:\WIN95\FONTS\SYMBOL.FON]
Symbolf [C:\WIN95\FONTS\SYMBOLF.FON]
Vgafix [C:\WIN95\FONTS\VGAFIX.FON]
Vgaem [C:\WIN95\FONTS\VGAEOM.FON]
Vgasys [C:\WIN95\FONTS\VGASYS.FON]
Marlett [C:\WIN95\FONTS\MARLETT.TTF]
Symbol [C:\WIN95\FONTS\SYMBOL.TTF]
Linedraw [C:\WIN95\FONTS\SLINEDRAW.TTF]
Alger [C:\WIN95\FONTS\ALGER.TTF]
Arlrdbd [C:\WIN95\FONTS\ARLRDBD.TTF]
Bookosb [C:\WIN95\FONTS\BOOKOSB.TTF]
Bragga [C:\WIN95\FONTS\BRAGGA.TTF]
Britanic [C:\WIN95\FONTS\BRITANIC.TTF]
Brushsci [C:\WIN95\FONTS\BRUSHSCI.TTF]
Colonna [C:\WIN95\FONTS\COLONNA.TTF]
Desdemon [C:\WIN95\FONTS\IDESDEMON.TTF]
Fhitt [C:\WIN95\FONTS\FHITL.TTF]
Gothic [C:\WIN95\FONTS\GOTHIC.TTF]
Impact [C:\WIN95\FONTS\IMPACT.TTF]
Kino [C:\WIN95\FONTS\KINO.TTF]
Latinwd [C:\WIN95\FONTS\LATINWD.TTF]
Maturasc [C:\WIN95\FONTS\MATURASC.TTF]
Playbill [C:\WIN95\FONTS\PLAYBILL.TTF]
Chklist.ms [C:\WIN95\FONTS\CHKLIST.MS]
Comic [C:\WIN95\FONTS\COMIC.TTF]

```

Running DD on the Desktop or the Network Neighborhood and specifying the `-r` switch can be time consuming. It may be quite a while before the program is finished working. In the meantime, thousands of lines may appear on the screen, from the Desktop all the way to the last file in the deepest directory of your hard drive. You can end the output any time you want by pressing **Ctrl Break**.

To view the program's output at anytime, simply redirect the output to a file. Then use a text editor to view the file later. To redirect the program's output, just add `>file` to the call after the parameters. Use the filename of your choice for file.

If you want to see what the Shell Namespace really contains, you'll soon come to like DD. For example, here's what you get when you call DD to view the contents of the fonts folder:

The listing for DD.C is shorter than you might at first believe. Much of the functionality is taken from the ENUMFOLD.C help module. You'll find both source files together with the MAK file in the `\WIN95\DD` directory on the companion CD-ROM.

You'll find the following program(s) on the companion CD-ROM



DD.C (C listing)

Setting the display name

You cannot set the attributes of ShellFolder objects through the IShellFolder interface. However, this is not true for the display name. It's possible to change the display name using the SetNameOf method. Therefore, you can change the names of files and directories on the hard drive. You can even change server names and the names for the virtual elements, such as those of the Desktop. However, this requires that the SFGAO_CANRENAME attribute of the object be set (see IShellFolder::GetAttributesOf). Only then is it possible to rename the display name by using SetNameOf.

The method expects more parameters than you might think at first. A new PIDL must be generated from the old one right away. The new display name must also occur in the item identifier as the representative of an object.

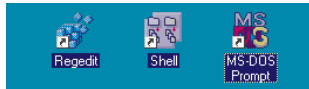
```
HRESULT SetNameOf(                                     // sets the name of an object
    IShellFolder *This,
    HWND hwndOwner,                                     // window handle of parent
    LPCITEMIDLIST pidl,                                 // address of item ID list of object
    LPCOLESTR lpszName, // pointer to UNICODE C string with new name
    DWORD dwReserved,                                   // 0
    LPITEMIDLIST *ppidlOut // address of a Var for new pointer item
                          // ID list
);
```

Therefore, the address of a PIDL variable must be specified in ppidlOut along with specifying the PIDL of the object to be renamed in the pidl parameter. ppidlOut is where the method writes the new PIDL. As usual, the memory for the item identifier list was allocated through the iMalloc interface. Therefore, the caller will have to release it later. By the way, the method automatically performs this operation for the entered PIDL, if it was allocated in the same way. If you aren't interested in a new PIDL, simply specify NULL for ppidlOut. Keep in mind that the new name is expected as UNICODE. As a result, ANSI strings must be converted to UNICODE strings using the MultiByteToWideChar() API function before you can pass them to the SetNameOf method.

Shortcuts And The IShellLink-Interface

Another important innovation of Windows 95 is *shortcuts*. Shortcuts are references with whose help you can create icons for the same file in as many folders as you wish. They can all refer to the same file. Clicking the shortcut will start the program to which the shortcut is pointing. Shortcuts are similar to the entries in the groups of the Program Manager under Windows 3. However, shortcuts can do more. They can point to all items that appear on the Windows 95 Desktop. These items include folders, network drives, Control Panel and much more. If a shortcut refers to a (data) file, its server application will be searched for based on the filename extension. The application starts up and loads the file as a document.

*Shortcuts are
recognized by
their icon*



You can always recognize shortcuts by the little arrow in the lower left corner of the icon. Shortcuts will quickly prove to be extremely practical, for example when several people are working with documents on one server. Creating shortcuts to the document is simple. First, place them on the Desktop. Then use them to access the document easily without having to memorize the UNC name of the server and the directory.

However, shortcuts are not an invention of Microsoft. Other operating systems have been using them for a long time. UNIX calls its system shortcuts *aliases*. What is new about shortcuts is the way they have been integrated into the system. Shortcuts can now, for example, be passed along without difficulties. Shortcuts are implemented in Windows 95 as independent OLE objects of the ShellLink class. So, they support some OLE interfaces through which they can be embedded in OLE containers. You can even send shortcuts as part of an E mail message.

Returning to the example of a workgroup: The project leader has finished the description of a new project and placed it on the server. She sends each member E mail to this effect with a shortcut to the document as a linked object. The team members

simply drag & drop the shortcut to their Desktop and now have access to the desired document at any time. It doesn't get any easier than that. New/Shortcut commands are available to users in several areas on the Desktop that lets them create a shortcut to an object. From the inside this functionality is available using the ShellLink object and its IShellLink interface. An application can use this interface to create new shortcuts, set their destinations and other features that make up shortcuts.

Attributes of a shortcut

Two items must be kept separate concerning shortcuts. Otherwise, you'll be confused immediately. One is the file of the object to which the shortcut is pointing. This is the shortcut destination, or the item that the shortcut represents. The other is the file in which a shortcut is saved. Naturally, you must put shortcuts somewhere otherwise they will be lost when you switch off the computer. Shortcuts are saved in files with the .LNK extension. They normally require less than 300 bytes. The 300 bytes are used for saving the shortcut destination and some additional attributes. The location of such a file determines where the user sees it. After all, an abbreviation is only useful if you know its beginning and the location of the .LNK file is the starting point for the abbreviation of the file to which the shortcut refers.

Place the shortcut in the Desktop directory so it's visible on the Desktop. Use the SHGetSpecialFolderLocation(CSIDL_DESKTOPDIRECTORY) function to find this directory. Then call SHGetPathFromIDist(). To have the shortcut appear in a folder, you must get the location of the folder on the hard drive. Then place the LNK file there. You can avoid this trouble if you only want the shortcut to appear in a specific hard drive directory. Place it where you want it. Besides their destination and storage location, shortcuts have the following attributes:

- Description of the shortcut
- Command line arguments for the call
- Display mode of the application window
- The icon under which the shortcut appears
- Working directory of the object
- A hotkey for direct call

Accessing shortcuts through the IShellLink interface

Use the methods of the IShellLink interface to set and query all these properties. IShellLink is the central interface for accessing shortcuts, although it's not the only one. The IPersistFile interface also implements the ShellLink class, and is used to write and load shortcut files (LNK files). Most methods of IShellLink are used for accessing the various properties of a shortcut, with a pair of Get... and Set... methods available for each property. The only exception is Resolve, which maintains the link between a shortcut and its destination.

Methods of IShellLink			
IShellLink	Task	IShellLink	Task
GetArguments	Displays the setting for the command line.	SetIDList	Sets the list of item identifiers.
SetArguments	Sets the command line.	GetPath	Supplies the path and filename of the shortcut.
GetDescription	Displays the description.	SetPath	Sets the path and filename of the shortcut.
SetDescription	Sets the description.	GetShowCmd	Supplies the setting for displaying the application window.
GetHotKey	Supplies the set hotkey.	SetShowCmd	Sets the display of the application window.
SetHotkey	Selects the hotkey.	GetWorkingDirectory	Reads the setting for the working directory.
GetIconLocation	Names the origin of the set icon.	SetWorkingDirectory	Sets the working directory.
SetIconLocation	Sets the icon under which the shortcut appears on the screen.	Resolve	Enables the link to the file to which the shortcut is pointing.
GetIDList	Supplies the list with the item identifiers	SetRelativePath	Sets the relative path between the shortcut and its destination.

Life cycle of a shortcut

You create a shortcut like other generic OLE objects by using `CoCreateInstance()`, an OLE API function. That's why you need the class and interface ID of `ShellLink` or `IShellLink`. They're provided by two variables that are provided by `SHELL32.DLL`. One is `CLSID_ShellLink` for the class ID. The second variable is `IID_IShellLink` for the interface ID. If the function is successful, you get a pointer to a new shortcut object when you call the specified interface variables. The new object instance is still uninitialized and exists only in memory. However, you can use the interface pointer to call the various methods for setting the properties of the shortcut, etc.

```
IShellLink *isl;

if (SUCCEEDED(CoCreateInstance(&CLSID_ShellLink,
                               NULL,
                               CLSCTX_INPROC_SERVER,
                               &IID_IShellLink,
                               &isl))
{
    printf("ShellLink object received\n");

    // now you can call IShellLink methods
    isl->vtable->SetDescription( isl,  "My wonderful shortcut\n");
    isl->vtable->SetPath( isl,  äC:\\DOS\\START.BAT");

    // additional calls to interface methods

    // don't forget to release the interface again
    isl->vtable->release(isl);
}
```

This occurs first in memory and not in a file. The shortcut must be placed in a file before it's visible to the user. The file can be stored right on the Desktop, in a Desktop folder or in any directory on the hard drive.

However, you won't find the required Save operation in the `IShellLink` interface. It's part of the standardized `IPersistFile` interface, which is also implemented by the `ShellLink` object. The `IPersistFile` interface is given preference because it is the standard interface for loading and saving objects to files. Each object that this interface offers to a client makes it especially easy for the parent object instances to handle different types of objects according to an identical system. As a result, one of the central ideas of object oriented programming called *polymorphy* is demonstrated here. You don't have to know the contents and the structure of an object. All you need to know is that it has a Load and Save function and how to call it. The object takes care of the rest.

To get to the `IPersistFile` interface within a `ShellLink` object, call the `QueryInterface` method specifying the interface ID from the external variable `IID_IPersistFile`. Call the two Load and Save methods of the interface using the returned interface pointer. Then you can save or load the `ShellLink` object. As a result, you can initialize a previously created, yet uninitialized instance of `ShellLink` by loading an existing shortcut. The other way around, you can use the various `Set...` methods to initialize a new instance to write it to disk, making the new shortcut persistent. The program code for this appears in the next section.

Use `DeleteFile()` to delete a file in which a shortcut is stored. It's not a function from an OLE interface or an OLE API function but a standard function from the Win32 API. If the object was on the Desktop or was visible in a folder it will automatically disappear.

Setting and displaying shortcut attributes

The simplest destination to which a shortcut points is a file somewhere on the hard drive, the local network or wherever a UNC filename will be adequate. After opening or generating a ShellLink object, use the SetPath method to set the path and filename of the file to which the shortcut refers.

```
HRESULT SetPath (          // Sets the name and path of the shortcut destination
    ShellLink *This,
    LPCSTR     pszFile      // Pointer to string with new path and filename
)
```

In the opposite direction, use the GetPath method to get the destination of a shortcut, i.e., the file to which it refers. For this purpose, the method expects a buffer in which it enters the path and filename. It also expects the address of a structure of the WIN32_FIND_DATA type. The method returns information about the shortcut file itself in this structure, for example its attributes and the date of its last modification. This data structure is defined in WINBASE.H and is normally used by the two API functions FindFirstFile() and FindNextFile() to browse through directories. In a manner of speaking, IShellLink::GetPath borrows this structure.

```
HRESULT GetPath(           // supplies path of the shortcut destination
    ShellLink *This,
    LPSTR      pszFile,     // pointer to buffer for receiving the shortcut
                        // destination
    int        cchMaxPath,  // size of buffer in bytes
    WIN32_FIND_DATA *pfd,   // pointer to WIN32_FIND_DATA type variable
    DWORD      fFlags       // UNC or 8.3 (one of the SLGP constants)
) ;
```

Use the fFlags parameter to determine which type of filename is returned. You can choose between the long (UNC) filename (SLGP_UNCPRIORITY constant) or the short 8.3 name (SLGP_SHORTPATH constant).

When using this method, keep in mind the query of the shortcut destination does not guarantee that the destination is still located in the specified location on the drive. The file could have been deleted, moved or renamed in the interim. To make sure the shortcut destination is where the shortcut believes it is, call the Resolve method before IShellLink::GetPath.

Command line arguments

If the shortcut destination is an executable file, you may want to specify command line arguments to influence the call of the program. This is one of the advantages of the shortcut principle: It lets you create several icons without much work. You can use these icons to start a program in different configurations.

Use SetArguments() to set the command line arguments, and use GetArguments() to display them. These are two very simple interface methods.

```
HRESULT SetArguments (     // sets command line arguments for shortcut destination
    ShellLink *This,
    LPCSTR     pszArgs      // pointer to C string with command line
) ;
HRESULT GetArguments (     // reads the command line arguments of shortcut
                        // destination
    ShellLink *This,
    LPSTR      pszArgs,     // pointer to buffer for command line
    int        cchMaxPath   // length of buffer
) ;
```

Description

You can also give a description to a shortcut. However, this information will not be displayed to the user (not even through the Property page). So, the description can only be of service to programs that edit shortcuts. The two interface methods `SetDescription()` and `GetDescription()` make this possible:

```
HRESULT SetDescription(          // sets description
    ShellLink *This,
    LPCSTR     pszName          // pointer to C string with new description
) ;
HRESULT GetDescription(          // displays description
    ShellLink *This,
    LPSTR      pszName,         // pointer to buffer for receiving description
    int        cchMaxName       // length of buffer
) ;
```

Working directory

When setting the working directory, you're specifying the directory that is selected as the current directory after starting an application (shortcut destination). If the program loads files without an explicit path specification, it automatically searches for them in the working directory and loads them from there.

```
HRESULT SetWorkingDirectory(     // sets working directory
    ShellLink *This,
    LPCSTR     pszDir           // pointer to C string with new working directory
) ;
HRESULT GetWorkingDirectory (    // reads working directory of shortcut
                                // destination
    ShellLink *This,
    LPSTR      pszDir,          // pointer to buffer for working directory
    int        cchMaxPath       // length of buffer
) ;
```

Shortcut hotkeys

Shortcuts support hotkeys, which allow you to start an application by pressing a key combination. However, you can only do this if the shortcut is directly on the Desktop. Hotkeys are described by a `WORD` whose low byte receives the virtual key code as one of the `VK_...` constants (such as `VK_F2` or `VK_HOME`). The shift keys that must be pressed in combination with the virtual key to execute the shortcut (or its destination) are set in the high byte. The following constants are available for making settings. They can be combined with each other:

Constants for setting the shift key for a shortcut hotkey	
Constant	Meaning
<code>HOTKEYF_ALT</code>	Alt key
<code>HOTKEYF_CONTROL</code>	Control key
<code>HOTKEYF_EXT</code>	Extended key (only found on new Microsoft Natural Keyboards)
<code>HOTKEYF_SHIFT</code>	Shift key

When you call `SetHotkey()`, the selected hotkey is passed directly as `WORD`. The selected hotkey is passed as the address of a `WORD` variable when you call `GetHotkey()`. The method fills the variable with the current hotkey.

```
HRESULT SetWorkingDirectory(     // sets working directory
    ShellLink *This,
```

```

        LPCSTR    pszDir    // pointer to C string with new working directory
    ) ;
    HRESULT GetWorkingDirectory (          // reads working directory of shortcut
                                          // destination

        ShellLink *This,
        LPCTSTR   pszDir,                // pointer to buffer for working directory
        int       cchMaxPath              // length of buffer
    ) ;

```

Show state of application window

If executing the shortcut leads to the display of an application window, use the `SetShowCmd` method to set the initial show state of this window. Choose whether to have the window appear minimized, maximized, normal or hidden. Ultimately, this is the parameter that the shortcut specifies in the internal start of the shortcut destination using `CreateProcess()` in the `STARTUPINFO.wShowWindow` parameter. The application to be started includes this parameter in its call to `ShowWindow()`, which is why both methods operate with the flags of `ShowWindow()`. Examples of these flags are `SW_NORMAL`, `SW_MINIMIZE`, `SW_MAXIMIZE` etc.

This flag is passed directly in the call to the `SetShowCmd` method. However, with `GetShowCmd` you must specify the address of an `int` variable, in which the function returns the flag currently set.

```

    HRESULT SetWorkingDirectory(          // sets working directory
        ShellLink *This,
        LPCSTR    pszDir    // pointer to C string with new working directory
    ) ;
    HRESULT GetWorkingDirectory (          // reads working directory of shortcut
                                          // destination

        ShellLink *This,
        LPCTSTR   pszDir,                // pointer to buffer for working directory
        int       cchMaxPath              // length of buffer
    ) ;

```

Shortcut icon

The creator of a shortcut has the opportunity to determine the appearance of the shortcut as the displayed icon. You can take the icon from any EXE or DLL file, whose name and path must be called. An index number is also available so you can address more than just the first icon within the file. The index refers to the order of the icon within the file. The “link overlay arrow”, which appears in the lower left corner of the icon to indicate the purpose of the icon, cannot be suppressed. The Shell automatically adds this arrow. In the call to `SetIconLocation` for setting the icon, the path is specified as a string pointer. The index is specified directly. With `GetIconLocation` you must specify a buffer’s size and address, so the method can return the current icon path. The buffer should contain `MAX_PATH` bytes so it can also receive long filenames.

```

    HRESULT SetIconLocation (          // sets the path for the shortcut icon
        ShellLink *This,
        LPCSTR    pszIconPath,        // pointer to C string with path and filename
        int       iIcon                // index of the icon
    ) ;
    HRESULT GetIconLocation (          // gets path for shortcut icon
        ShellLink *This,
        LPCTSTR   pszIconPath,        // pointer to buffer that receives
                                          // path & filename
        int       cchIconPath,         // length of buffer
        int       *piIcon              // pointer to Int that receives index
    ) ;

```

The relative path

The relative path between the shortcut file and its destination is the only ShellLink property that you cannot retrieve, but only set. The relative path is optional information and is intended to help you find the destination of a shortcut if it has been moved. If the file was moved as part of a complete directory tree, often the shortcut is moved with it because the shortcut and the file are established within the same tree. This results in the relative path between the shortcut file (.LNK file) and its destination remaining the same, even if both files are now elsewhere. For example, if there is a shortcut file called “Letter to Miller.LNK” in the \letters directory and the shortcut destination, the “miller01.doc” file, is located in the \letters\personal directory, the relative path between the shortcut destination and the shortcut file will be personal\miller01.doc. If the entire letters directory including all of its subdirectories is moved to \\BigServer\WalterZabel, afterwards the shortcut file “Letter to Miller.LNK” will be in \\BigServer\WalterZabel\letters, while its destination will be in \\BigServer\WalterZabel\letters\personal\miller01.doc. The relative path stays the same.

Besides the obligatory this pointer, the function expects a pointer to the C string with the relative path between the shortcut file and its destination. As usual, you get an HRESULT as your result.

```
HRESULT SetRelativePath (           // sets relative path between shortcut
                                // and destination
    ShellLink *This,
    LPCSTR     pszPathRel,         // pointer to C string with relative path
    DWORD      dwReserved         // reserved, 0
) ;
```

Updating the link between a shortcut and its destination

We can consider many scenarios where files are moved without the relative path between the shortcut file and its destination remaining the same. So, why bother setting the relative path if it doesn’t do you any good? This uncovers a basic problem with shortcuts, also expressed in the Resolve method. The problem is the link between the shortcut and its destination exists in only one direction. The shortcut knows exactly which file it is pointing to, but the file doesn’t know which shortcuts are pointing to it. Because files haven’t been treated as active objects to this point, the file system doesn’t have the option of automatically finding the shortcuts pointing to a file when it moves a file. Yet this is exactly what is required so the shortcut destination can be reset in the appropriate shortcuts. Only an object oriented file system, as is to be expected in one of the future versions of Windows/NT, will make this possible.

That’s why you now must always assume that a file to which a shortcut is pointing is no longer in the same location that the shortcut thinks. The file didn’t simply vanish into thin air on its own...somebody had to move, delete or rename it. Before you can retrieve a shortcut to access the shortcut destination through the path contained within, make certain it’s still where you think it is. The Resolve method in the IShellLink interface is in charge of this task.

If Resolve doesn’t find the shortcut destination in the expected location, it attempts to track down the file, thus restoring the integrity of the shortcut. Among others, it uses the relative path, which is still the easiest method for finding the destination. If it still doesn’t find the file, it searches the directory of the original destination for a file with the same attributes as the file. A file with the same date, same size and flag indicates that the file stayed where it was but has been renamed. If this attempt also fails, Resolve searches all the subdirectories of the shortcut destination in case the file has simply gone a few levels deeper in the directory hierarchy.

If this also doesn’t work, a dialog box appears prompting the user to enter the location of the file. If Resolve doesn’t succeed in reestablishing the link in this way, the call to Resolve fails. This means you’re no longer able to reach the file from the shortcut.

```
HRESULT Resolve ( // updates the link between shortcut and its destination
    ShellLink *This,
    HWND       hwnd,           // window handle in case dialog box is displayed
    DWORD      fFlags          // SLR_... flags
) ;
```

Along with the obligatory `This` pointer, a window is expected as the first parameter of the window handle. This window acts as the parent for a dialog box that may have to be displayed if the shortcut cannot be resolved. If the caller doesn't have a window, he can specify a `NULL`. Also, you can specify a combination of different SLR constants in the `Flags` parameter that determine the execution of the method.

Constants for the call to the <code>Resolve</code> method	
Constant	Meaning
<code>SLR_UPDATE</code>	Specifying this flag means that the file not only will be tracked down, but the new path will also be saved in the shortcut file immediately.
<code>SLR_ANY_MATCH</code>	If the shortcut destination cannot be found, the user gets a chance to specify the new location of the file in a dialog box.
<code>SLR_NO_UI</code>	When this flag is specified, the method doesn't display a dialog box to the user if the shortcut destination cannot be found. Instead, it returns with an error.

When you specify `SLR_NO_UI`, you can set a timeout duration in milliseconds in the hi word of the flags `DWORD` parameter. After this time passes, the search for the shortcut destination will end. Specifying a 0 (only the `SLR_NO_UI` constant) sets the maximum timeout duration to 3 seconds.

Specifying a timeout always makes sense when you are on large networks and want to limit the amount of time the `Resolve` method spends searching drives from the Network Neighborhood. If the `Resolve` method returns with a success code `S_OK`, you'll find the valid path and filename of the shortcut destination in the `Path` property of the shortcut (`GetPath` method).

Other objects besides files

Shortcuts don't just point to files but can refer to all the items in the Shell Namespace. This includes the Control Panel and the program groups as well as the Network Neighborhood. You don't reference these objects using strings with path and filenames. Instead, reference these objects through the item identifier lists that we've already mentioned. If you get the PIDL of an object, whether by calling `SHGetSpecialFolderLocation()` or through the `IShellFolder` interface, set the object as the destination of the shortcut by using the `SetIDList` method. To do this, all you need is the PIDL that you pass to the method along with the `This` pointer.

```
HRESULT Resolve ( // updates the link between shortcut and its destination
    ShellLink *This,
    HWND      hwnd,          // window handle in case dialog box is displayed
    DWORD      fFlags          // SLR... flags
) ;
```

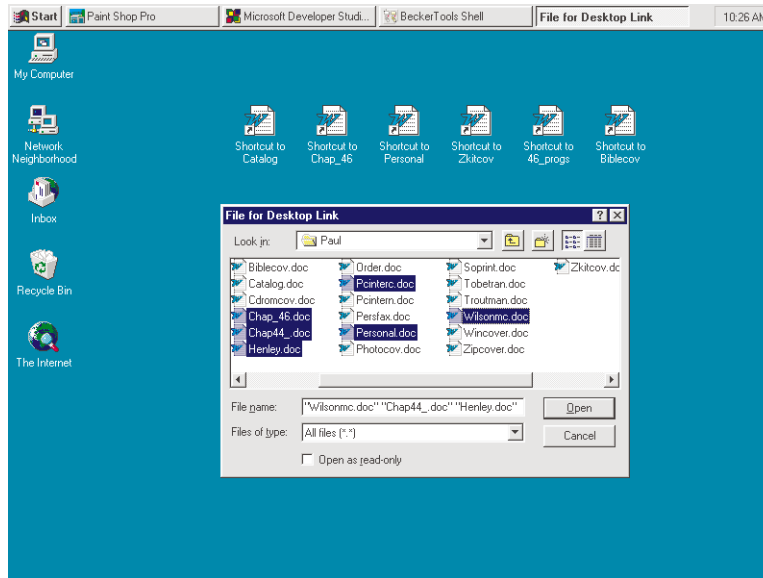
There is also a method called `GetIDList` that you can use to get the PIDL of a loaded shortcut. In the prompt for the current item identifier list, you must supply an `LPTITEMIDLIST` type variable and specify its address. The method places the address of the current item identifier list in there. You cannot make any changes to this list. If you wish to use it for your purposes in the program code, you'll have to make a copy.

```
HRESULT GetIDList( // supplies shortcut destination as pointer to
                  // item ID list
    ShellLink *This,
    LPTITEMIDLIST *ppidl // pointer to PIDL pointer that receives
                        // the address
) ;
```

Sample program

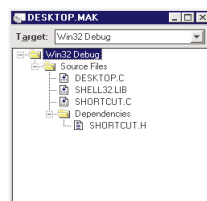
DESKTOP.C is an example of working with shortcuts. This program and its companion files, DESKTOP.MAK and SHORTCUT.C, are on the companion CD-ROM in the \WIN95\DESKTOP directory. DESKTOP gives users the opportunity to select one or more files from a default dialog box for opening files. The program then creates shortcuts to the selected files and places them in the hard drive directory of the Desktop. That's why the shortcuts appear immediately on the user's Desktop.

First the files are selected, then their shortcuts appear on the Desktop



DESKTOP.C uses the CreateShortcut() function from the SHORTCUT.C help module to create the shortcuts and place them on the hard drive. It's DESKTOP.C's task to display the common dialog, display the files created by the user and create them using CreateShortcut(). However, before this, the program determines the location of the Desktop directory through SHGetSpecialFolderLocation(). It then converts it to a path name with SHGetPathFromIDList() so the shortcuts to be created will be saved in the directory of the Desktop immediately.

Files in the DESKTOP.MAK project



You'll find the following program(s) on the companion CD-ROM



DESKTOP.C (C listing)

Creating Programs And Program Groups

In conjunction with the Shell function SHGetSpecialFolderLocation(), we showed the program groups of the Desktop are established as directories below the Windows directories, in the WINDOWS\STARTMENU\PROGRAMS directory. All the directories contained within represent program groups, their directories are subgroups, and we could go on forever. The names of the directories represent the names of the program groups. Within the different program group directories, files and shortcuts are also established along with subgroups as subdirectories. These files and shortcuts appear on the screen as the items of the program group.

You'll usually work with shortcuts because it's better during installation to copy programs together with all their files to a separate directory. So you don't place the EXE file in the hard drive directory of the program group, but only a shortcut to it. When the user clicks the appropriate entry on the screen, the shortcut starts the EXE file.

Creating new program groups

Therefore, to create a new program group and fit the desired program items in it, a program must:

1. Get the PIDL of the hard drive directory for program groups using SHGetSpecialFolderLocation(CSIDL_PROGRAMS).
2. By using the SHGetPathFromIDList() function, convert the returned PIDL to a path and directory name.
3. Create a subdirectory with the name of the new program group within the program group directory by using the Win32 API function, CreateDirectory().
4. Within this new directory, create the required shortcuts to the programs and documents of the application.
5. Inform the Shell of the changes using SHChangeNotify(), so the changes are reflected on the screen.

The following partial code shows this procedure.

```
#include <windows.h>
#include <shlobj.h>

void CreateNewGroup(void)
{
    LPITEMIDLIST pidl;
    char szProgramPath[ MAX_PATH ];
    char szNewGroup[ MAX_PATH ];

    if( SUCCEEDED( SHGetSpecialFolderLocation( NULL,
                                                CSIDL_PROGRAMS, &
                                                pidl ) ) )
    {
        // Get path of program group directory. _____
        SHGetPathFromIDList( pidl, szProgramPath );

        // Make path for new group _____
        lstrcpy( szNewGroup, szProgramPath );
        lstrcat( szNewGroup, "\\NewGroup" );

        SetLastError( 0 );

        // Create new group or directory _____
        CreateDirectory( szNewGroup, NULL );

        if( ( GetLastError() == 0 ) ||
            ( GetLastError() == ERROR_ALREADY_EXISTS ) )
        {
            // Shortcuts would be created here in the familiar way
            // .
            // .
            // .

            // Announce changes to main program directory (!) _____
            SHChangeNotify( SHCNE_UPDATEDIR,
                           SHCNF_FLUSH | SHCNF_PATH,
                           (LPVOID)szProgramPath,
```

```
        0 );  
    }  
    CoTaskMemFree( (LPVOID)pidl );  
}  
}
```

The old DDE Interface of the Program Manager

If you have installed program groups using software in Windows 3, you're familiar with the DDE interface of the Program Manager. Windows 95 supports this interface as well, so you can use it easily. However, we recommend the interfaces described in this chapter for new developments because DDE is now being viewed as a technology that has outlived its usefulness. Future versions of Windows will no longer actively support DDE.

Naturally, you don't have to use Start/Programs to create the new program groups. Instead, use the IShellFolder interface to browse the Namespace below the program to find a subgroup or a subsubgroup underneath which you want to create the new program group. When you browse, you must attach the item identifiers of the different folders to each other to build an item identifier list as an absolute path from the Desktop to the desired folder. Then use SHGetPathFromIDList() to convert this item identifier list to the name of the hard drive directory and create the desired program group within it.

Sample program

PROGRAM.C shows how creating program groups and their shortcuts occurs on a large scale. This program could be part of an installation program if you wanted the user to be able to select the desired program group for displaying program files at the end of installation. A tree view appears in a dialog box for the user, reflecting the current structure of the program groups and their subgroups.

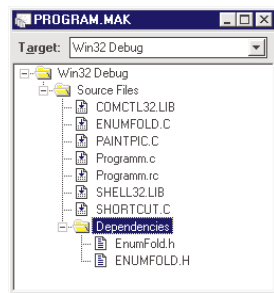
After the user selects one of the displayed program groups, he can enter the name of the new, yet to be created program group in the input box under the tree view. The program group is then generated. Also, three shortcuts are created in it for demonstration purposes. All three shortcuts point to the EXE file of PROGRAM. Choose the Properties command in the Context menu to display their contents.

The program is represented by the PROGRAM.MAK make file. The program uses two help modules, ENUMFOLD.C and SHORTCUT.C. The program uses these help modules to display the Shell Namespace, paying special attention to the program groups.

This occurs in the FillTreeWithGroups() function, which uses the EnumFolders() function for this purpose. The enumeration begins with CSIDL_PROGRAMS, so that only the program groups and their contents in the Namespace are browsed. The EnumShellCallback() function is specified as a callback function. It receives the various objects in the browsed section of the Namespace and places them in the TreeView of the dialog box.

However, it only does this if the objects are folders, i.e., in this case, other groups. Shortcuts and file objects within the various program groups are not placed in TreeView. The icons for displaying the program groups are obtained through the SHGetFileInfo() function.

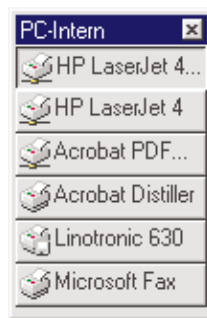
*Files in the
PROGRAM.MAK
project*



Setting The Printer

The NameSpace of the Shell also displays the available printers in the Printers folder, as the folder appears on the Desktop. The CSIDL_PRINTER constant represents this folder, which you can access through SHGetSpecialFolderLocation(). This allows a program to display the names and driver files of the available printers. However, adding printers and setting the default printer are quite another thing. That's what we'll talk about in this section. We'll write a program that lets you switch between various default printers quickly and easily. You'll see this is nothing like using the Control Panel or some other Printer dialog box.

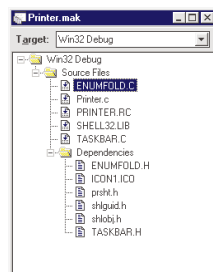
*PRINTER.C
attaches itself to
the TNA and
displays a dialog
box for setting
the default
printer when you
click it*



PRINTER.C, located in the \WIN95\PRINT directory of the companion CD-ROM, makes this possible. It also shows how to use the Taskbar Notification Area. A small icon is anchored on the Taskbar. A dialog box appears for selecting the printer when you click this icon. One click is all it takes to switch between laser printer and color deskjet printer.

Because the program isn't just responsible for printer settings, but also manages the TNA and the printer selection window, the listing is "a bit" longer than originally planned. However, to make up for it, the program has become a complete system utility for Windows 95. It's ideal for using in similar projects.

*Files in
PRINTER.MAK
project*



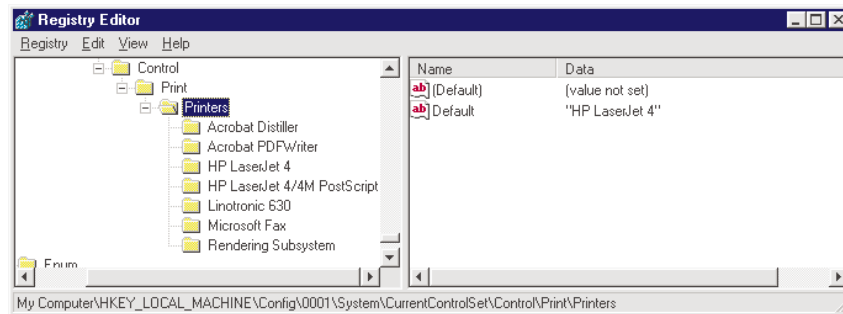
PRINTER.C contains the majority of the functionality module. The program also uses the TASKBAR.C module to control the TNA. Also, the program uses the ENUMFOLD.C module to browse the Namespace. First, we'll look at the functions used for determining the available printers, querying the current default printer as well as for setting a new default printer.

Querying and setting the existing printers

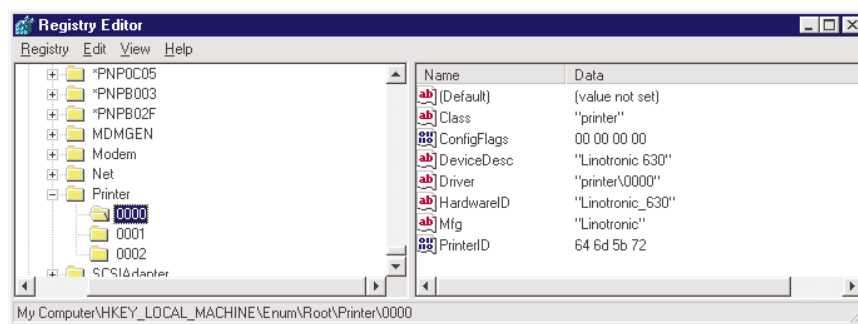
Since we know the Registry is the central location for all types of settings, it's safe to assume the available printers and their drivers are also stored here. You

might also think it's safe to assume that you can also specify the default printer by setting the corresponding Registry entry. However, only the first assumption is true. As the next three illustrations show, entries for the available printers and their drivers are located in different areas of the Registry. It's just that you can't use them to set the printer. That is, you can set the printer here, but the system couldn't care less. The next time you print, the system routes the output back to the old default printer unless you explicitly specify a different printer.

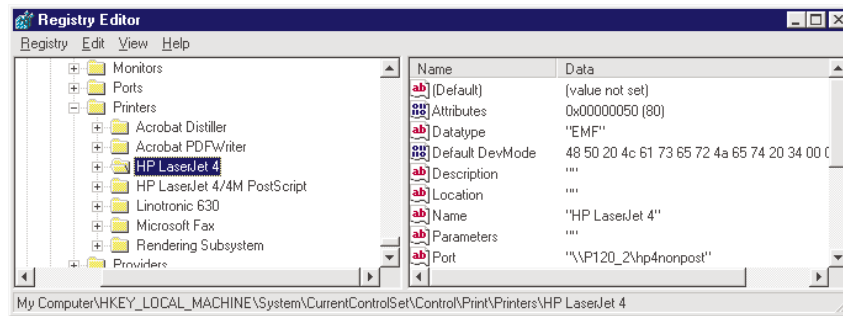
Available printers in the Registry, including the current printer (the current printer can't be set here)



Is the current printer listed here?



Is the current printer listed here?



You'll find the solution to this puzzle in the WIN.INI file. It wasn't supposed to have such an important role in Windows95, but exceptions prove the rule. Not only is the current printer recorded in this file, you also set the current printer in this file. In our research, this was the only option we could find for explicitly setting the default printer. You won't find a corresponding API function.

It appears that the system gets the default printer from the WIN.INI but maps it to the corresponding entries of the Registry. Once you change the entry in the WIN.INI and the system learns of this change, the Registry will also change its printer entry. It just won't work the other way around. Apparently, you must use the WIN.INI to define the default printer.

Within the WIN.INI you will find the appropriate entry in the [windows] section under device=.

```
[windows]
load=
run=
NullPort=None
device=TI microLaser Pro 600,HPCL5MS,\\Mitil\tips
```

First, specify the name of the printer, then a comma, followed by the name of the printer driver and finally the connection to the PC. Normally this will be LPT1: or some other printer port. It's a UNC name in our example because the default printer is connected to a printer server on the network. To determine the default printer, you must retrieve this entry from the WIN.INI. You need to change this entry in the WIN.INI to set the printer. Our sample program includes two functions (SetStdPrinter() and GetStdPrinter()) to perform these tasks.

GetStdPrinter() is much shorter because the INI entry just needs to be retrieved by the API GetProfileString() function and cut off before the first comma so you get the printer name only.

With SetStdPrinter(), on the other hand, you must create the complete entry for the INI file. This involves the name of the printer, name of the printer driver and the port. Use the GetPrinterDriver() and GetPrinter() functions to get this information from the Win32 API. Then combine it in a string and use WriteProfileString() to write it to the WIN.INI file. Remember to use Broadcast to send the WM_WININICHANGE message to all the windows. This will make the other processes aware of the changed settings in the WIN.INI. This is also how the system determines about the changes so it can begin inspecting the WIN.INI. In so doing, the system will run into the modified device= entry and accepts the specified printer as the new default printer.

All that remains is to display the available printers. This occurs through the PRINTER folder in the Shell Namespace using the EnumFolders() function from the module with the same name. Call EnumFolders() in the FillWindowsWithPrinters() function. It's the task of this function to create a button in the printer selection window for each available printer. Most of this work is performed by the callback function, which EnumFolders() calls for each browsed entry in the PRINTER folder. In our example, the callback function is the EnumShellCallback() function of the PRINTER.C module.

Structure of the printer selection dialog box

Beyond querying for and setting the default printer, the rest of the program consists largely of GDI and USER calls. It begins in EnumShellCallback(), where a new Ownerdraw button is created within the printer selection dialog box for each printer. For its Control ID, the button gets the value of the ID_PRINTERBUTTONS constant plus the consecutive printer number. Data for each printer is placed in a global variable called g_Printers. This variable is an array with entries of the PRINTERBUTTONS type. This array is defined at the beginning by PRINTER.C. The individual entries receive the window handle of the button, the current location and size within the window as well as the handle of the icon and other information. However, the entries don't get the button label, because this is set when it is created with SetWindowText().

The buttons are drawn within the dialog procedure of their parent window in the PrinterWndProc() function. The other activities of the program are also coordinated here. These activities include the initial creation of the window and the clicking of one of the printer buttons and the accompanying switching of the default printer through a call to SetStdPrinter(). Also, when the user tries to resize the window, the WM_SIZE message is processed. If possible, the printer buttons will be redistributed and the window will be adapted to the desired size.

The CreateStdPrinterWindow() function creates the window for switching printers. We're getting close to the end of the listing now. All we have left are the TaskbarWndProc() function and the winmain() start function.

Control through the TNA

An invisible window is created first in WinMain(). It's then specified as the parent when the icon is added to the Taskbar Notification Area. As a result, its message procedure TaskbarWndProc() is called whenever there is a message for the application's TNA icon.

When the WM_CREATE message is received in this procedure, it calls CreateStdPrinterWindow() for the first time. This is done to create the window for printer selection with the individual printer buttons.

Its window handle is stored within the message procedure in a static local variable called hPrinter. That way, it is available if other messages come. This window will be displayed when the program icon in the TNA is double-clicked.

If the right mouse button is pressed over the icon, the function first creates the self-defined WM_CONTEXTMENU message to itself. When the message arrives, the function creates a menu that consists only of the "Remove" entry and is displayed as the Context menu.

When the user selects Remove, it signals the program to close its window and terminate.

You'll find the following program(s) on the companion CD-ROM



PRINTER.C (C listing)

Calling The Control Panel

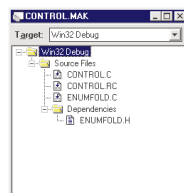
The Control Panel also appears as a virtual folder in the Shell Namespace. This makes it possible for a program to retrieve and start the control applets. Control applets are the control programs for the various system resources. For example, anyone who wants to guide the user from their application to configuration of the modem simply needs to call the appropriate applet. All the necessary information comes from the Shell Namespace. This includes the display name (e.g., Display or System) as well as a path and filename, which you can get from the PIDL by using SHGetPathFromItemID(). The display name represents the name of a CPL file which contains the code of the applet. CPL files are normal DLLs that export some predefined functions that allow them to identify themselves as control applets and can be used to call the applets.

To start an applet, call the Control Panel, i.e., the CONTROL.EXE program using ShellExecute() or CreateProcess(). Pass the name and path of the CPL file and then the display name of the desired applet in the command line. You need the display name because some CPL files have several control applets. That's how you identify which applet you need.

Sample program

CONTROL.C includes program code for calling a control applet. When you call this program, the Control Panel displays a dialog box with a list view control. The program loads the icons and names of the available control applets into this control. When the user double-clicks one of the icons, the applet starts up in the manner we've been describing.

*Files in the
CONTROL.MAK
project*



As with some of the earlier programs in this chapter, CONTROL.C also uses the ENUMFOLD.C help module with its EnumFolders() function to browse the Namespace. In this case, the function browses the contents of the virtual folder represented by the CSIDL_CONTROLS constant. The call to EnumFolders() occurs in the FillListWithControlApplets() function. It's called after the start of the program to load the list view within the dialog box with the names of the Control Panel and its icons.

The icons are taken from the system image list, where the Shell keeps the icons of the items appearing on the Desktop. That's why at the beginning of FillListWithControlApplets(), the handles of the system image list for the small (16*16) and the normal (32*32) icons are determined first. Then the handles are anchored in list view so the view takes the icons to be displayed from both image lists.

Only then does the call to EnumFolders() occur, with EnumShellCallback() being specified as the callback function. This function is in the same module that has responsibility for transferring the individual control applets to the list view of the dialog box. For this purpose, the function determines the icon index of the applet in relation to the system image list, inserting this together with the display name as a new entry in list view. Also, the new entry gets a pointer that refers to a buffer allocated within the function. The buffer receives the path name, which will be required later for the call to CONTROL.EXE.

This memory is released later within the message procedure called DialogProc(), when the program ends. Before this happens, however, the function makes certain the user can start control applets from the dialog box by double-clicking them or pressing the Run button. Then the StartControlApplet() function is called. It puts together the command line for calling CONTROL.EXE from the entry of the selected list view item and starts the program using ShellExecute().

Your programs could do it in the same way. It's possible, under the right circumstances, that it's unnecessary for the user to see the different control applets. Maybe you only want to call one particular applet after the user selects a menu or starts some other action. In any event, the ControlPanel program gets you started off on the right foot.

Dialog Boxes For Opening And Saving Files

To save the programmer unnecessary work in coding dialog boxes for opening and saving files and to standardize the appearance of these dialog boxes at the same time, Windows 95 includes the same GetOpenFileName() and GetSaveFileName() functions from Windows 3. Most of the dialog boxes for loading and saving files that you see in commercial programs are displayed using these two functions.

The old Win16 functions continue in Windows 95 in their usual DLL called COMMDLG.DLL. You will find the Win32 versions of these functions in a file called COMMDLG32.DLL. The Win32 versions of these functions have the same name but have been adapted to the 32 bit model and some of the enhanced features of Windows 95. You should use GetOpenFileName() and GetSaveFileName() to make displaying the corresponding dialog boxes as easy as possible. That's why we'll talk about these functions in this section.

GetOpenFileName() and GetSaveFileName()

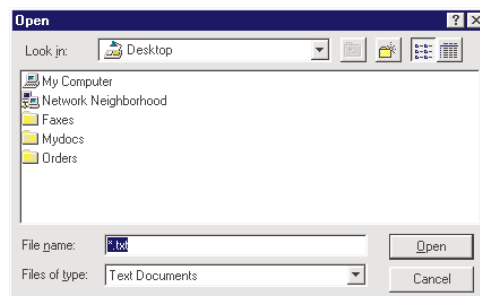
As the prototypes show, the two functions are identical in structure, and are controlled to a large extent by a structure of the OPENFILENAME type. The caller has to initialize this structure and pass it to the function through a pointer.

```

BOOL GetOpenFileName(                                // Display OpenFileName dialog box
    LPOPENFILENAME lpofn    // Pointer to variable of the OPENFILENAME type
);
BOOL GetSaveFileName(                                // Display SaveFileName dialog box
    LPOPENFILENAME lpsfn    // Pointer to variable of the OPENFILENAME type
);

```

*How the
common dialogs
for opening and
saving files
appear on the
screen*



In both cases, TRUE is returned as a function result when the user left the dialog box using Open. FALSE is returned when the user closed the dialog box with Cancel.

The OPENFILENAME structure determines largely how the function will operate. We'll talk about the various fields later. First, we'll talk about the capabilities of both functions, as reflected in the makeup of the data structure.

A series of string pointers to buffers created by the caller are expected. For example, you might have a string pointer to a buffer for receiving the

selected filename. If the function is to place data in this buffer, another parameter specifying the size of the buffer is expected. This prevents the function from writing beyond the buffer in memory. For example, a field called `nMaxCustFilter` is available for `lpstrCustomFilter` and a field called `nMaxFile` is available for `lpstrFile`.

When we describe the individual fields, we'll occasionally talk about *filters*. Filters are predefined file types in the combo box in the lower left corner of the dialog box. The caller can predefine a filter's contents, so the user can easily select specific file types using a predefined, or default file extension. We'll also be talking about *callbacks* and *templates*. These are advanced options of the two functions that you won't use very often because they intervene in the internal processes of the dialog box. We'll take a closer look at both of these options later in this chapter.

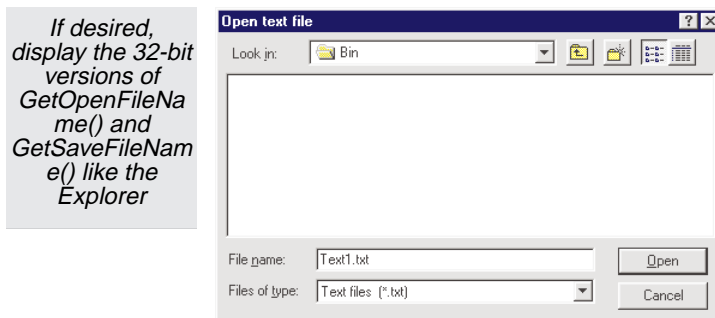
```
typedef struct {
    DWORD          lStructSize;           // Size of structure in bytes
    HWND           hwndOwner;             // Window handle of parent
    HINSTANCE      hInstance;            // Instance handle, only for templates
    LPCSTR         lpstrFilter;           // Pointer to string with display filter
    LPSTR          lpstrCustomFilter;     // Pointer to buffer for selected
                                           // filter
    DWORD          nMaxCustFilter;        // Size of this buffer in bytes
    DWORD          nFilterIndex;          // Pattern or description of user selected
                                           // filter
    LPSTR          lpstrFile;             // Pointer to buffer containing selected
                                           // filename
    DWORD          nMaxFile;              // Size of buffer to which lpstrFile is
                                           // pointing
    LPSTR          lpstrFileTitle;        // Buffer containing additional file&ext.
    DWORD          nMaxFileTitle;         // Size of buffer in lpstrFileTitle
    LPCSTR         lpstrInitialDir;       // Ptr to C-Str with initial directory
                                           // for display
    LPCSTR         lpstrTitle;            // Ptr to C-Str with title of dialog box
    DWORD          Flags;                 // Combination of OFN constants
    WORD           nFileOffset;           // Offset to filename in lpstrFile
    WORD           nFileExtension;        // Offset to extension in lpstrFile
    LPCSTR         lpstrDefExt;           // Pointer to C-Str with default extension
    LPARAM         lCustData;             // LPARAM value for callback function
    LPOFNHOOKPROC  lpfnHook;              // Pointer to callback function
    LPCSTR         lpTemplateName;        // Pointer to template name as C string
} OPENFILENAME, *LPOPENFILENAME;
```

The Flags

The `Flags` field plays a special role in controlling how the function works. The caller can specify any combination of OFN constants in this field. For example, OFN constants determine whether the user can enter path and filenames that don't even exist, set up a callback or not be allowed to select files that are Read-Only. Most of these flags were and are available to the Win16 versions of both functions as well. Some flags have been added for the Win32 version of the two functions, which relate to specific capabilities of Windows 95 and its Shell.

Constants for the OPENFILENAME.Flags field that are related to the new Win95 features	
Constant	Meaning
OFN_EXPLORER	Switches display of the dialog box from the typical appearance of Windows 3 to the style of the Explorer under Windows 95.
OFN_LONGNAMES	If this flag is not specified, only the old 8.3 filenames will be displayed. This only applies if the display hasn't been switched to Explorer style. The Explorer always displays the long filenames.
OFN_NOLONGNAMES	Counterpart to OFN_LONGNAMES.
OFN_NODEREFERENCELINKS	If the user selects a shortcut, the function returns the destination of the shortcut. In other words, the shortcut is automatically retrieved to get its destination. Specifying this flag causes the function to return the name and path of the shortcut file (the LNK file) instead.

The OFN_EXPLORER constant is definitely an important constant. Use it to give dialog boxes the look of the Explorer.



The following summarizes the remaining flags. First, is the OFN_ALLOWMULTISELECT flag. It lets the user select multiple files in a directory. We'll tell you more about that in the next section. The different flags list the possibilities the functions offer to the caller for influencing file selection and the display of drives and directories.

Default flags for the OPENFILENAME.Flags field	
Constant	Meaning
OFN_ALLOWMULTISELECT	Allows the user to select multiple files within a directory. See next section, "Selecting multiple files".
OFN_PATHMUSTEXIST	The specified path on the selected directory must exist, otherwise the user can only exit the dialog box by pressing Cancel.
OFN_FILEMUSTEXIST	The specified file must exist, otherwise the user can only exit the dialog box by pressing Cancel. Automatically includes OFN_PATHMUSTEXIST.
OFN_CREATEPROMPT	Prompts user to create a new file when he/she specifies the name of a file that does not exist. If the user confirms the prompt, a new empty file is created with the filename. Setting this flag automatically sets the OFN_PATHMUSTEXIST and OFN_FILEMUSTEXIST flags.
OFN_HIDEREADONLY	Hides the check box that lets users toggle on or off the display of Read-Only files.
OFN_NOCHANGEDIR	When a user moves from directory to directory, they not only select the displayed directory within the dialog box, they also make it the current directory. When you specify this flag, the function switches back to the original current drive after the end of the selection.

Default flags for the OPENFILENAME.Flags field (continued)	
Constant	Meaning
OFN_NONETWORKBUTTON	Specifying this flag hides the network button, which otherwise lets the user connect to other network drives.
OFN_NOREADONLYRETURN	The user can only exit the dialog box using Cancel if they specify a file that is read-only or is located in a write-protected directory.
OFN_OVERWRITEPROMPT	This flag is only important in the call to GetSaveFileName() and is intended to prevent the user from accidentally overwriting an existing file. When the user specifies a file that already exists, the function asks him whether he really wants to overwrite the file before exiting the dialog box. Only when the user confirms the prompt does the function return to the caller. Without this flag an existing file will be destroyed without a prompt.
OFN_SHAREAWARE	When this flag is specified, a call to GetOpenFileName() will return successfully with the selected filename even when the file is on a network drive and it has been determined during a text access that the file may not be read.
OFN_SHOWHELP	Displays the Help button within the dialog box. However, this requires that the caller has specified his window handle in OPENFILENAME.hWnd, and not something like NULL.
OFN_ENABLEHOOK	This is the prerequisite for the function calling the callback function of the caller from the OPENFILENAME.lpfnHook field during the display of the dialog box.
OFN_ENABLETEMPLATE	Lets you add custom controls to the dialog box of the Common Dialogs. (see section "Expanding the Common Dialogs")

The remaining parameters for the OPENFILENAME structure

However, these flags aren't the only fields in OPENFILENAME that determine the execution of the function. The following explains the remaining parameters.

lStructSize

Size of the structure, initialize with sizeof(OPENFILENAME).

hwndOwner

Window handle of parent so the parent cannot be enabled by the user while the dialog box is displayed.

hInstance

Instance handle of the application or DLL, in which the template for expanding the Common Dialog is located. Only required when OFN_ENABLETEMPLATE is specified in Flags.

lpstrFilter

Pointer to several logically connected C strings. The NUL byte at the end of a string separates these strings from each other. C strings define the filters. Filters are the entries that appear in the File Type combo box in the lower left corner of the dialog box. Examples of filters would be "Textfile *.txt" or "Word document *.doc". The user uses filters to select the files to be displayed from the current directory. In this way the caller can give the user hints about the files that a particular program is able to process.

Within the buffer that is pointing to pstrFilter, each filter must be described by two C strings. The first C string specifies the type of file, such as "Text file", "Make file" and the like. You can combine several such extensions in a string by separating each entry with a semicolon (,*.A;*.B;*.C").

You can specify one or more of these pairs. To display the end of the list, add two blank strings, i.e., NUL bytes, to the last valid pair.

lpstrCustomFilter

This parameter points to a buffer in which the `CommonDialog` function places the user input regarding the filter when the dialog box closes. Along with one of the predefined filters, the user also has the option of entering a filter by hand. To keep this entry and have it appear the next time you display the dialog box, specify a pointer for `lpstrCustomFilter` to a buffer prepared for this purpose. The buffer must have room for a minimum of 40 characters.

nMaxCustFilter

Specifies the size of the buffer to which `lpstrCustomFilter` points. The size will be 0 if you specify `NULL` for `lpstrCustomFilter`.

nFilterIndex

Before the call of the function, the caller can specify a filter here with which the display is to be started. The filename of the corresponding filter appears in the "Filename" edit/text box, and only files matching the filter will be displayed from the directory. The filters are counted starting with 1. The value 1 enables the first filter, 2 enables the second filter, etc. If you specify 0, no filter will be set. Then the function will operate by the value specified in `lpstrCustomFilter`. If a string was called in `lpstrCustomFilter`, (`lpstrCustomFilter <> NULL`), the initial filter is taken from this string and the matching filename is displayed in the file selection box. However, if `NULL` is also specified in `lpstrCustomFilter`, the function searches `lpstrFilter` as a final alternative. If this field is also `NULL`, there is no filename and the file selection box will be empty. However, if there is a string in `lpstrFilter`, the first filter in the string will automatically be used as the default filename.

After the function call, you will find the number of the filter selected by the user, provided a filter has been selected. Otherwise the value in this field will be 0.

lpstrFile

This pointer refers to the buffer where the function places the filename selected by the user. The filename is a C string. This C string includes the filename, drive and path specification. The buffer must have at least `MAX_PATH` characters of room available. Before the call, the buffer must be loaded either with a blank string (first byte `NUL`) or with a string presented to the user as the default.

nMaxFile

The number of characters that will fit in the buffer to which `lpstrFile` is pointing.

lpstrFileTitle

The function places the selected filename here, including the extension. However, the drive and path specifications are omitted unless you specify `NULL`.

nMaxFileTitle

The number of characters that fit in the buffer to which `lpstrFileTitle` is pointing. If you specify a `NULL` for `lpstrFileTitle`, use a value of 0 here as well.

lpstrInitialDir

To have the display begin with a specific directory, specify a pointer to a C string here. This C string is where you have placed the path and drive specification of the desired directory before the function call. To display the current directory on the current path, specify `NULL` here.

lpstrTitle

Specify a string pointer here whose C string is displayed as the title of dialog box. Specify `NULL` if you don't want a title for the dialog box. In this case, "Save as" or "Load" will appear.

Flags

A combination of the OFN constants listed above.

nFileOffset

After the function call, you will find an offset to the beginning of the filename within the returned drive, path and filename string in *lpstrFile*. For example, if *lpstrFile* has „d:\cdrom\docs\thefile.doc“, you will find a value of 14 in *nFileOffset*. This indicates the offset position of „thefile.doc“ within the entire string. So you can use this field to get the selected filename from the entire string.

nFileExtension

Follow the same procedure with *nFileExtension* that you do with *nFileOffset*. It's a method for quickly filtering the filename extension from the string.

lpstrDefExt

Using this pointer, the caller can define a default file extension before the call. That way, when the user doesn't specify a file extension, this default extension will be added to the filename.

lCustData

This parameter can be of use if you specify a callback function in *lpfnHook* for linking to the events within a dialog box. This parameter represents a long parameter (LPARAM), which is passed to the callback function when a WM_INITDIALOG message is sent in its *lparam*. As a result, a connection between the caller of the function and the code of the called callback function can be established.

lpfnHook

The actual callback pointer, which refers to the function used by the caller to intervene in the message processing of the default dialog box. For more information on defining this type of callback function, see the “Expanding the Common Dialog Boxes” section. The function pays attention to this field only when the OFN_ENABLEHOOK constant is set in *Flags*. Otherwise the contents of this field are ignored.

lpTemplateName

Here you can define a pointer to a C string with the name of a dialog box or its ID. For more information, see the “Expanding the Common Dialog boxes” section.

Selecting multiple files

So users can select multiple files if you specify the OFN_MULTISELECT flag. Then the selected files are returned in the buffer specified through OPENFILENAME.*lpstrFile* in a special format. Because all the selected files must come from the same directory, you name the selected directory, along with its drive, first. This is followed by a space, and then the first selected filename, including its extension. Another space follows the extension which is followed by the next filename, then another space, etc. The last filename is not followed by a space, but rather by a NUL byte that ends the string. Naturally, the buffer specified in OPENFILENAME.*lpstrFile* must be large enough to ensure that all the filenames will be returned perfectly. This is especially true if you specify OFN_EXPLORER or OFN_LONGNAMES, meaning the filenames that are returned will be long ones.

The structure of the buffer remains the same by specifying OFN_EXPLORER. However, the individual filenames and the preceding path won't be separated by spaces. They're instead separated by a NUL byte. After all, long filenames can also contain spaces, which would really mess up the interpretation of the strings. Therefore, the filenames are packed in a separate string. After the last filename there is only a blank string, starting with a NUL byte. So at the end of the buffer you will find two NUL bytes. One NUL byte ends the last filename and the other one represents a blank string. It marks the end of the list.

Expanding the Common Dialog boxes

When you display Explorer style dialog boxes (OFN_EXPLORER Flag), you have the option of expanding the dialog boxes by adding controls. For example, you could add buttons, or checkboxes.

Before the function call, set the OFN_ENABLETEMPLATE constant in OPENFILENAME.Flags. Also, you must load the two OPENFILENAME.hInstance and OPENFILENAME.lpTemplateName fields so the function can load the desired dialog resource and insert the controls contained within into the common dialog box. The function first determines which module contains the dialog resource by using hInstance. The dialog resource could be in the current program or in one of the DLLs loaded by the program. Use lpTemplateName to specify a pointer to a C string containing the name of the dialog resource. However, naming resources through strings is an option that is seldom used. Therefore, you'll probably want to specify the dialog ID of the dialog resource instead of the string. Use the MAKEINTRESOURCE macro (from WINUSER.H) for this purpose. It receives the dialog ID and casts it in the required string pointer. The common dialog function can tell the difference between the two types of specifications because IDs are never larger than 64K while the location of a string in memory doesn't begin before 4 Meg (the loading address for programs).

Specifying the dialog resource causes the dialog controls contained within to be inserted into the common dialog box. However, this isn't very useful. Their contents and the selection of the user are lost when you close the dialog box. So, you need an option for responding directly to the selection of the user within the dialog box. This is the purpose of the *callback*. You can specify the callback by using the OPENFILENAME.lpfnHook field. It gives the caller the opportunity to participate in the message flow within the dialog box to be able to respond to user events. For example, the caller could respond to the clicking of a checkbox to note the current contents of the checkbox in a global variable so the contents are still present. This can happen even after the dialog box has been closed and the caller returns from GetOpenFileName() or GetSaveFileName()

However, to ensure that the callback function is called, you must set the OFN_ENABLEHOOK constant in OPENFILENAME.flags. Without this flag, even the setting of OPENFILENAME.lpfnHook doesn't do you much good. The function pointer specified in OPENFILENAME.lpfnHook must be of the LPOFNHOOKPROC type. It's defined in the following manner:

```
UINT (APIENTRY *LPOFNHOOKPROC) (HWND, UINT, WPARAM, LPARAM);
```

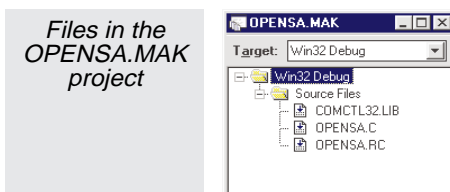
The function receives the usual parameters in a message procedure as arguments. This can be, for example, first the window handle of the window, then the actual message code and finally wParam and lParam with additional information about the message. The callback function assumes the role of a hook. Use this "hook" to listen in on the message traffic without processing the individual messages. However, you can remove messages from the message stream, so they won't be processed by the dialog box any more. This is determined by the return value of the callback function. If you specify 0, the message will continue to be processed in the normal way. A return value not equal to 0 means the message will be ignored.

Use the OPENFILENAME.lCustData parameters to establish a connection between the caller of GetOpenFileName() or GetSaveFileName() and the callback function. The next program shows how this is done.

Sample programs

OPENSA.C is an example of using and expanding the common dialog box for opening and saving files. This program and its MAK file are located on the companion CD-ROM in the \WIN95\OPENSA directory. So the call to the common dialog

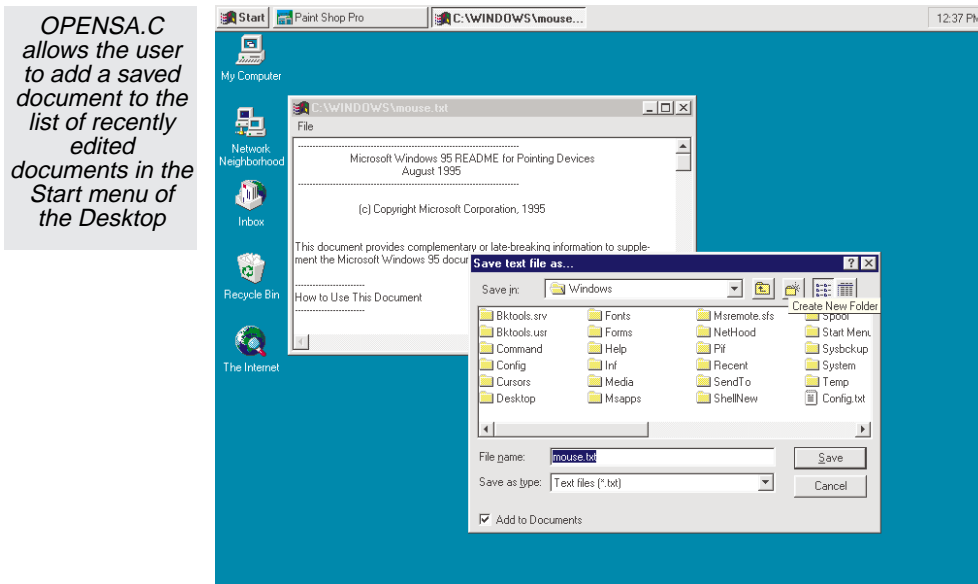
boxes has a meaningful framework, the program represents a small text editor. This text editor has a multiline text box to display and edit the loaded files. The File menu gives the user access to the Open, Save and Save as commands. These commands use the two functions, GetOpenFileName() and GetSaveFileName(), to let the user select the desired files. The dialog boxes are displayed in Explorer style. With Save and Save as the common dialog box is expanded by a separate template that comes from the resource file of the project (OPENSA.RC) and is stored there under the ID IDD_SAVEEXTENSION. You will find the



code for displaying this dialog box within the OPENSA.C module in the SaveEditToFile() function. The SaveDlgExtension() function is employed as a callback function.

SaveDlgExtension() monitors the user input in the “Add to Documents” checkbox. It’s added to the common dialog box in the template. This box gives the user the chance to add the saved file to the list of recently used documents in the Start menu of the Desktop. For OpenSaveDlg.C only a call to the Shell API function SHAddToRecentDocs() is required. This function creates a shortcut to the document and places it in the appropriate place on the hard drive. It then appears in the Start menu of the Desktop.

However, the call to SHAddToRecentDocs() doesn’t occur within the callback. Instead, it occurs after the dialog box is closed and the return to the caller, which is the SaveEditToFile() function in this case. Therefore, the function has to work hand in hand with the callback function so the user’s selection in the “add to documents” checkbox doesn’t become lost when the common dialog box is closed. Naturally, the two could use a global variable for this purpose, but we chose a more flexible solution here for demonstration purposes.



SaveEditToFile() declares a local variable called bAddToRecentDocs that the callback function is to load with true or false, depending on the selection of the user. For this purpose, the function must first get the address of this variable. That is why the variable is passed to the GetSaveFileName() function in the OPENFILENAME.lCustData parameter. As a result, when it receives a WM_INITDIALOG message, the callback function gets this parameter as lParam of the message. The function saves it in the static local variable, pbAddToRecentDocs, so that it is available when other messages are received. In this way the callback function gets permanent access to the bAddToRecentDocs variable of the SaveEditToFile() function. Naturally, this is only until this function is terminated and its local variables end. However, the callback won’t be called anymore either because the common dialog has to be closed before the API GetSaveFileName() function returns to the SaveEditToFile() function.

Upon receipt of the WM_INITDIALOG message, the “Add to Documents” checkbox is initialized with the IDC_ADDTORECENTDOCS ID using the pointer to the variable. It is initialized with the value that SaveEditToFile() placed in its local variable.

After initialization has been completed from the view of the callback function, the WM_COMMAND messages referring to the IDC_ADDTORECENTDOCS control take on the greatest

You'll find the following program(s) on the companion CD-ROM



OPENSA.C (C listing)

significance. When the user changes the setting of the checkbox, the new selection is reflected in the local variable of `SaveEditToFile()`. In this way, the last valid setting remains persistent even after the common dialog box is closed. It can then be used by `SaveEditToFile()` as a basis for deciding whether `SHAddToRecentDocs()` is called for the file.

Drag and Drop From The Desktop

Applications located near the Desktop always risk having the user accidentally drop files from the Desktop or its folders onto the application window of the application using drag and drop. Many applications can't do anything with such files. This is why the Shell doesn't even let users drop a file on such an application. On the other hand, applications that support dragging and dropping of files have to explicitly declare their willingness for this to the Shell. Only then can users drag files over an application and drop them.

This section talks about the different API functions that are involved in the idea of drag and drop and show how applications can get files using this method. For example, an API function is available for displaying data, executing specific operations on files or simply saving them in the framework of the current document.

We'll concentrate on one of two methods for achieving drag and drop. This method uses Windows messages and therefore has been available since Windows 3. The only difference is that it was the File Manager from where you could take the files under Windows 3. However, it's the Desktop with its icons and folders or the Explorer that serve as the starting point for drag and drop under Windows 95. Along with this method through the Win32 API, the OLE specification also contains a drag and drop interface designed especially for OLE objects. However, this interface is totally different in operation and can only be implemented with a great deal more time and effort. That's why we concentrate on the traditional mechanism, which you can use to make your application support drag and drop with only a few steps.

Working with the drag functions

The call to the `DragAccept()` function starts all the work, and should occur during the initialization of an application. This informs the Shell that the application can serve as the target of a drag and drop operation. Before termination of the application, another call must be made to this function to undo the application's support for drag and drop. The various drag functions are included in `SHELL32.DLL`. They're defined in the `SHELLAPI.H` Include file.

The drag functions in the Win32 API	
Function	Task
<code>DragAccept</code>	Indicates that an application is a drop target or prevents any more files from being dragged over an application window.
<code>DragQueryFiles</code>	Returns the filenames of the dragged files.
<code>DragQueryPoint</code>	Returns the window coordinate over which the files were dragged.
<code>DragFinish</code>	Indicates to the Shell that all the dragged files have been accepted and the drop operation is thus complete.

The `fAccept` flag is used to indicate willingness to receive files. The `DragAcceptFiles()` function expects the window handle of the application window as its first parameter.

```
VOID DragAcceptFiles(           // Toggles acceptance of files using drag & drop
                           // on/off
    HWND hWnd,                 // Window handle of application
    BOOL fAccept                // TRUE = Drag&Drop allowed, FALSE = not
);
```

That the process or instance handle of the application is not expected here is your first clue that drag & drop is managed at window level and not at the level of the process. An application determines about the placement of files over its application window using drag and drop from a message in the window's message procedure. The message is called `WM_DROPFILES`.

When this message is received, you will find a value of the **HDROP** type in the **wparam** message parameter. It represents a handle for the query of the dragged/dropped files. Specify this value in a call to the **DragQueryFile()** function to determine the names of the dragged files.

```
UINT DragQueryFile (           // Retrieve filenames of dragged files
    HDROP    hdrop,           // Drop handle from WM_DROPFILES.wparam
    UNIT     iFile,           // Number of dragged file in our query
    LPTSTR   lpszFile         // Pointer to buffer for filename as C string
);
```

You can only get one filename with each call of this function. This is why you must call the function several times in a row in cases where multiple files have been dragged. The best way to determine exactly how many times you must call the function is to call **DragQueryFile()** specifying a value of **0xFFFFFFFF** for the **iFile** parameter. The functions don't determine the filenames with this value. They instead return the number of dragged files in the function result. That way you know how often you still need to call **DragQueryFile()** to get the names of the dragged files.

Start with a value of 0 in **iFile** to get the first filename. Continue incrementing **iFile** with each call until all the filenames have been determined. Specify the name of a buffer in **lpszFile** that has to have room for **MAX_PATH** characters to be able to hold long filenames. Be careful here because the function doesn't get the size of the buffer from a separate parameter.

However, you can determine the buffer size in advance by calling the function with the desired value in **iFile**, but specifying **NULL** for **lpszFile**. In this case the function returns only the length of the filename. If you have a buffer of the appropriate length (plus one byte for the closing NUL byte) ready, nothing will go wrong in the call with a real buffer address. In this case, you get the number of characters the function copied to the buffer as a function result. Therefore, you're getting the length of the filename again. Therefore, you don't receive the contents of the files, but only their names. To get to the contents, you must open the files in the conventional ways and then read them.

If you also want to determine the position at which the user dropped the files, for example to establish whether the user placed them in a specific area or over a particular dialog control, use the **DragQueryPoint()** function. Unlike **DragQueryFile()**, you only need to call this function once. This is true regardless of the number of dragged files because all the files were dropped over the same point.

```
BOOL DragQueryPoint (           // Query coordinate of drag area
    HDROP    hdrop,           // Drop handle from WM_DROPFILES.wparam
    LPPPOINT lppt             // Address of an LPPPOINT structure that is to receive
                             // the coordinate
);
```

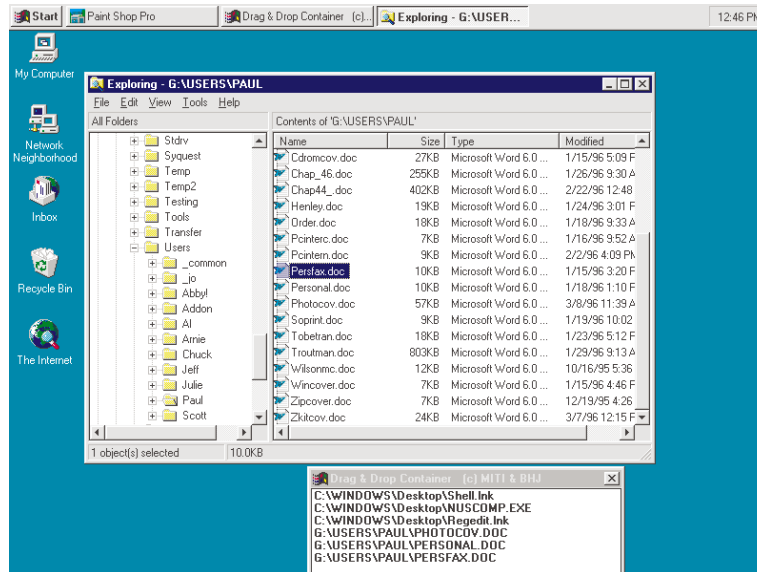
The exact point is determined through a structure of the **POINT** type. The caller must specify its address in the **lppt** parameter. You get **TRUE** back as a function result if the drag occurred over the client area of the caller. Finally, call the **DragFinish()** function after all the files have been received. The application uses **DragFinish()** to display the end of the drag operation. This gives the Shell the opportunity to free up the resources that were reserved internally for the drop operation. Above all, this includes the memory required to note the names of the different files. Therefore, the function must be passed the drop handle from the original **WM_DROPFILES** message.

```
BOOL DragFinish (              // Concludes drop operation from the view of an
                               // application
    HDROP    hdrop,           // Drop handle from WM_DROPFILES.wparam
);
```

Sample program

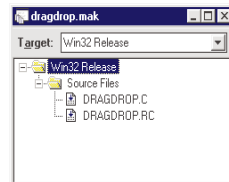
DRAGDROP.C shows you how to receive files in your program using drag and drop. **DRAGDROP.C** is a small program that displays a dialog box (**IDD_DIALOG**) at startup with the **IDC_DRAGLIST** listbox. This is where the program places the names of all the files that it receives using drag and drop from the Explorer, the Desktop or any other application.

DRAGDROP.C demonstrates receiving files using drag and drop



This is possible because, in `DlgProc()`, the message procedure of the dialog box, upon receiving the `WM_INITDIALOG` message, the `DragAcceptFiles()` function enables drag and drop. Upon receipt of a `WM_DROPFILES` message, the dropped files are retrieved using `DragQueryFile()` and their names are added to the listbox. The program proves that little work is involved in including drag and drop operations in your programs.

Files in the DRAGDROP.MAK project



You'll find the following program(s) on the companion CD-ROM



DRAGDROP.C (C listing)

Taskbar Notification Area (TNA)

The Taskbar is also part of the Desktop, which creates and manages it. All programs, except one, are usually denied access to the Taskbar. The one exception is the Taskbar Notification Area (TNA). This is the area at the right or bottom border of the Taskbar, next to the clock, where the tiny icons appear. For example, you might see an icon for the printer here. Applications can add icons here. When you click on the icon, the program starts running. Many programs are likely to use this versatile option. Use the Taskbar Notification Area if you want to do the following:

- Keep your program active in the background
- Display the status or your program permanently
- The user to call the program without much hassle and without a program icon on the Desktop

Accessing the TNA

Accessing the TNA is quite easy. Simply use the `Shell_NotifyIcon()` function to send one of three messages to the Taskbar. `Shell_NotifyIcon()` is declared in `SHELLAPI.H`. The messages are represented by NIM constants, representing one of three basic operations. In the opposite direction, the Taskbar sends an application all the mouse messages triggered by user actions within the icon area. The mouse message could be a mouse movement (`WM_MOUSEMOVE`) or the clicking of the icon

(WM_LBUTTONDOWN or WM_RBUTTONDOWN). This is the backward channel through which the application can respond to user events. We'll return to this in the next section. First, let's look at the messages that an application sends to the Taskbar through Shell_NotifyIcon().

However, a message by itself is not enough to convey all the necessary information to the function. So, in the call to Shell_NotifyIcon(), the NIM message is always accompanied by a structure of the NOTIFYICONDATA type. This structure supplies the Taskbar with the information that it requires for executing the desired operation.

```

BOOL Shell_NotifyIcon(
    DWORD          dwMessage,          // one of the NIM constants
    NOTIFYICONDATA lpData              // pointer to NOTIFYICONDATA variable
);

```

When you create the structure, initialize the first field of the structure, cbSize, with sizeof(NOTIFYICONDATA). The function uses this field to make certain that the individual fields really are present before it accesses them.

```

typedef struct {
    DWORD   cbSize;          // sizeof(NOTIFYICONDATA)
    HWND    hWnd;            // window handle for callback messages
    UINT    uID;              // ID randomly selected by caller
    UINT    uFlags;           // NIF constants, indicating the occupied fields
    UINT    uCallbackMessage; // another ID randomly selected by the caller
    HICON    hIcon;           // handle of icon being notified
    CHAR    szTip[64];        // C string with max. 64 characters for tooltip
} NOTIFYICONDATA, *PNOTIFYICONDATA;

```

Use hWnd to set the window that is to receive the callback messages from the Taskbar. Specify the window handle of your application window here or another window that deals with the corresponding messages in its message procedure. uID helps the Taskbar distinguish several icons of a program. If you call the function several times for various icons, enter different uID values in each of the NOTIFYICONDATA structures so you can tell the icons apart. Specify in uFlags which fields of the structure you want to have filled with information for the Taskbar by combining different NIF constants. That way the function will recognize which attributes it needs to retrieve from the structure and set for the icon.

Messages to the Taskbar Notification Area	
Constant	Meaning
NIF_ICON	There's a valid icon handle in hIcon.
NIF_TIP	The tooltip to be displayed is in szTip.
NIF_MESSAGE	There is a message in uCallbackMessage.

This also gives us the meaning of the three remaining fields. Pass the handle of the icon, which must be 16 x 16 points, in hIcon. In szTip you can place a C string with a maximum length of 63 characters (plus the terminating NUL byte). This text will be displayed when the user leaves the mouse pointer over the icon for a short time, but doesn't click the icon. Finally, you pass the message to be sent to the window of the caller when there are mouse events for the icon in uCallbackMessage. So, you can set your message code to which the custom message procedure will later respond.

Life cycle of an icon in the Taskbar Notification Area

First, call Shell_NotifyIcon() with an NIM_ADD message to create a new icon. You'll be able to tell whether it worked by the function result. To modify one of the different attributes later, use the initialized NOTIFYICONDATA structure again, with the NIM_MODIFY message. In this way, you can modify the icon, the tooltip text and even the message that you will

Messages to the Taskbar Notification Area	
Constant	Meaning
NIM_ADD	Add new icon to TNA
NIM_MODIFY	Change attributes of a displayed icon
NIM_DELETE	Remove an icon

get from the Taskbar later for mouse events. You don't have to change all three attributes at once. By not combining all the NIF constants in the uFlags field, you prevent the function from changing all the attributes.

Sending an NIM_DELETE message marks the end of a cycle. Use the NOTIFYICONDATA structure again that we've been using. The icon is removed from the TNA. No more messages will be passed on to the window of the caller from that point.

TASKBAR.C help module

To eliminate as much work as possible in dealing with the TNA, we wrote a help module for you called TASKBAR.C. It contains all the necessary routines for creating a new icon in the TNA, for setting its properties, as well as deleting the icon.

You'll find the following program(s) on the companion CD-ROM



TASKBAR.C (C listing)

Internet Shortcuts On The Desktop

Microsoft has given us an example of the power of shortcuts in a small DLL called URL.DLL. You can generate shortcuts to HTML pages in the WEB, FTP server and all other items of the Internet through this DLL. Address the shortcuts through URLs (Uniform Resource Locator). When you click the shortcut, a connection to the Internet is established and the appropriate item is displayed, for example, a page in the World Wide Web. Since putting this type of shortcut on the Desktop is no problem, this is a great opportunity for all types of programs designed to guide the user to a given location on the Internet.

However, this will only work if the Internet access has already been created on the machine. Also, there must be a protocol server established in the Registry for the Internet protocol you are using (FTP, HTTP etc.). Internet shortcuts have a dialog box called a Property Sheet for setting their properties. In the Property Sheet you can set not only the URL but also all the other parameters that determine the display of the URL.

Accessing Internet shortcuts

Internet shortcuts represent objects with a custom class ID and an OLE interface called IUniformResourceLocator. You can create and set Internet shortcuts through this interface. Also, Internet shortcuts implement the IPersistFile interface so you can save shortcuts to and load them from files. The standard interface for shortcuts, IShellLink, is also implemented. However, not all the methods of the interface are supported. Only those methods that make sense with an Internet shortcut are supported.

The C interface for InternetShortcut objects is defined in the INTSHCUT.H Include file. You'll also find the declarations of two external variables with the class ID and the interface ID of the InternetShortcut objects. Feed the addresses of these variables to the OLE API CoCreateInstance() function, to get a new InternetShortcut object and an interface pointer to its IUniformResourceLocator interface. Not only can you access the methods of the IUniformResourceLocator interface through this interface pointer, you can also get pointers to the supported IShellLink and IPersistFile interfaces using the QueryInterface() method contained within. Afterwards release this interface and the original interface pointer to the IUniformResourceLocator interface through the Release method, which is part of every interface. However, for now let's talk about the IUniformResourceLocator interface and its methods.

The methods of IUniformResourceLocator	
IUniformResourceLocator	Task
SetURL	Sets the desired URL (the target of the shortcut)
GetURL	Retrieves the set URL
InvokeCommand	Displays the URL, or executes another command

The first two methods access the most important attribute of an Internet shortcut, the URL, to which the Internet shortcut points. As with all OLE methods, they return an HRESULT. It can be queried using the SUCCEEDED and ERROR macros.

```
HRESULT SetURL(
    IUniformResourceLocator * This,
```

```

PCSTR  pcszURL,                // pointer to C string with URL
DWORD  dwInFlags                // IURL flags
);

```

To set a URL through the SetURL method, along with the mandatory this pointer all you need is the address of a C string with the URL (pcszUrl), and a flag. The flag comes into play whenever the URL doesn't contain a protocol specification, i.e., doesn't begin with "HTTP:" or "FTP:". There are two constants available for the flag:

IURL_SETURL_FL_GUESS_PROTOCOL

If no specification of an Internet protocol is in the URL, an attempt is made to determine the appropriate protocol from the URL. For example, if an HTML page is in the URL, you know that you're working with the HTTP protocol of the World Wide Web. If you don't specify this flag, no attempt will be made to find the correct protocol if it is missing.

IURL_SETURL_FL_USE_DEFAULT_PROTOCOL

HTTP, which is the default protocol, is used if the specification of an Internet protocol is missing.

ALL_IURL_SETURL_FLAGS

Combines the two flags.

You don't need quite so many parameters to determine an Internet shortcut's URL. In this case, along with this, it's enough to specify the address of a variable in which the function places a pointer to a C string with the desired URL. The caller must release the memory of this buffer when it is no longer needed. You can use the CoTaskMemFree() function from the OLE API for this purpose.

```

HRESULT GetURL(
    IUniformResourceLocator * This,
    PSTR *ppszURL           // Addr. of a var. in which the pointer to the buffer
                           // is placed
);

```

To initiate an Internet shortcut from a program, i.e., connect to the Internet and place the Internet item on the screen, use the InvokeCommand method. This is tantamount to the user's double-click on the shortcut.

```

HRESULT InvokeCommand(
    IUniformResourceLocator *This,
    PURLINVOKECOMMANDINFO  purlici           // pointer to URLCOMMANDINFO
);

```

The execution is determined by a variable of the URLCOMMANDINFO type, whose address the caller must specify in the purlici parameter. The structure is defined in INTSHCUT.H and is organized in the following manner:

```

typedef struct {
    DWORD  dwcbSize;                // sizeof(URLINVOKECOMMANDINFO )
    DWORD  dwFlags;                 // one of the IURL flags
    HWND   hwndParent;             // window handle of parent
    PCSTR  pcszVerb;               // pointer to string with verb names
} URLINVOKECOMMANDINFO;

```

The caller must initialize the dwcbSize field with the size of the structure, that is, with sizeof(URLINVOKECOMMANDINFO). In dwFlags you can specify a combination of the following flags:

IURL_INVOKECOMMAND_FL_ALLOW_UI

If this flag is missing, no dialog box will be displayed when a problem exists connecting to the net. An example of a typical problem is when no server is set up for that protocol or the Internet access is currently busy). Instead, an error will be returned immediately. However, if you set this flag, the user gets the chance to correct this problem in a dialog box. If it is not possible to correct the problem, the method returns with an error.

IURL_INVOKECOMMAND_FL_USE_DEFAULT_VERB

Specifying this flag calls the default verb (open).

ALL_IURL_INVOKECOMMAND_FLAGS

Combines the two flags, which are followed by the `hwndParent` parameter. It's to be loaded by the caller with the window handle of their application window. However, this only happens when you specify the `IURL_INVOKECOMMAND_FL_ALLOW_UI` flag because a parent window is then required for displaying dialog boxes. If you do not set this flag, specify 0 for `hwndParent`.

`pcszVerb` is only required if the `IURL_INVOKECOMMAND_FL_USE_DEFAULT_VERB` flag is not specified. In this case, the caller determines the action to be carried out on the Internet shortcut through a C string with the desired command. The address of this string is passed in `pcszVerb`. You can specify "open" or another verb that is executed in the context menu of an Internet shortcut, such as "print".

Loading and Saving Internet shortcuts

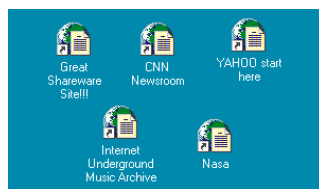
Loading and saving shortcuts are handled by the methods of the `IPersistFile` interface. If you don't specify a file extension when calling the Load and Save methods, the shortcut automatically gets the .URL extension. This extension is required to have the system recognize the Internet shortcut as an Internet shortcut and let it be processed by the appropriate server application (URL.DLL).

Automatically generating Internet shortcuts

The URL.C sample program demonstrates how to create shortcuts of a program and place them on the Desktop. We didn't want the user to have to enter the desired URLs so instead we preset five URLs. When you start the program, it creates five Internet shortcuts on your Desktop. Naturally, this is provided the corresponding Internet system files have been set up properly. If not, a dialog box informing you of this appears on the screen instead.

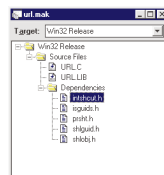
The `CreateInternetShortcut()` function is the main function of the following listing. It is called from the main program for each Internet shortcut to be created. It expects as arguments the target of the Shortcut (the URL) as well as the name and path of the file in which the Internet shortcut is to be placed.

The Internet shortcuts created on the Desktop by URL.C on different pages and servers in the Web



To have the shortcuts appear on the Desktop, the start function first uses `SHGetSpecialFolderLocation()` and `SHGetPathFromIDList()` to determine the location of the Desktop directory.

Files in the URL.MAK project



You'll find the following program(s) on the companion CD-ROM



URL.C (C listing)

The IShellLink Interface of an InternetShortcut object

As we've indicated, the InternetShortcut object also implements the IShellLink interface, although not all the methods. The only methods that have been implemented are those for setting and querying attributes that InternetShortcuts share with normal shortcuts. You can also call the nonimplemented functions, because the entry in the vTable is initialized with a meaningful code pointer. It's simply that this pointer points at nothing more than an inner function that only returns the E_NOTIMP error code.

In part, the implemented methods refer to the properties of the shortcut target (the URL), however, some of the methods concern the protocol server, i.e., the application that must be started to display the URL.

The methods of IShellLink as part of an InternetShortcut object	
IShellLink::	Task
SetPath/GetPath	Supplies or sets the URL as target of the shortcut. Corresponds to the call to IUniformResourceLocator::SetUrl or ::GetUrl().
SetDescription/GetDescription	Gets or sets the path of the InternetShortcut file (*.URL) on the hard drive.
SetIconLocation/GetIconLocati	Sets or gets the origin of the icon for the InternetShortcut.
SetShowCmd/GetShowCmd	Gets or sets the attribute that determines the display of the protocol handler's application window. You can specify one of the default constants like SW_NORMAL etc. here.
SetWorkingDirectory /GetWorkingDirectory	Sets / gets the setting for the working directory of the protocol handler. This is the file where the protocol handler puts the default files.
SetArgument/ GetArguments	not implemented
SetIDList /GetIDList	not implemented
SetHotkey/GetHotKey	not implemented
Resolve	not implemented
SetRelativePath	not implemented

One interesting option here is the possibility of giving the Internet shortcut a different icon. You cannot do this with the IUniformResourceLocator interface. Only by doing this can you give each Internet shortcut a distinctive appearance, making it stand out from all the other Internet shortcuts. You'll find a function called SetInternetShortcutIcon() in the URL.C program above to perform this task for you.

The ENUMFOLD.C help module

Several programs in this chapter use the ENUMFOLD.C module when working with PIDs and browsing the Namespace. ENUMFOLD has some functions that will save the caller much of this work.

PIDL help functions in the ENUMFOLD.C module	
Function	Task
pidlNext	Gets a pointer to the next item identifier within an item identifier list.
pidlGetSize	Gets the size of a complete item identifier list.
pidlGetNumOfItems	Gets the number of item identifiers within an item identifier list.
pidlCreate	Creates a buffer for receiving an item identifier list and initializes it, so that it only consists of an empty item identifier.
pidlConcat	Links one item identifier list to another one.
pidlClone	Creates an independent copy of a given item identifier list.

The listed PIDL functions help you work with PIDLs, for example, building item identifier lists from individual items or breaking down existing lists into their individual parts. These functions are a kind of “string function”, except they are for PIDLs.

In addition, the module has two other helpful functions: `ExtractStrRet()` and `EnumFolders()`. The first function handles the troublesome conversion of filename strings as they are returned by the `GetDisplayNameOf` method in a `STRRET` structure. You don’t need to worry about whether the string is returned as an offset, in Unicode or somewhere in the PIDL. Instead, you get the filenames specified in `STRRET` converted and in a separate buffer as an ANSI C string.

Other functions in the ENUMFOLD.C. module	
Function	Task
<code>ExtractStrRet</code>	Gets ANSI string from <code>STRRET</code> .
<code>EnumFolders</code>	Browses through all the objects in a folder and calls the callback function of the caller.

`EnumFolders()` is a function that is always needed for browsing the Shell Namespace or one of its folders. It gets its name from the module. By using this function you can browse through one of the folders, whose PIDL you can get by using the `SHGetSpecialFolderLocation()` function.

We’re talking about browsing the entire spectrum of folders and Desktop items that exist for a CSIDL constant. For example, this would include `CSIDL_DESKTOP` for the Desktop, as the root of the whole system or `CSIDL_FONTS` for the folder with the installed fonts.

```
typedef struct {
    DWORD dwcbSize;                // sizeof(URLINVOKECOMMANDINFO )
    DWORD dwFlags;                 // one of the IURL flags
    HWND hwndParent;              // window handle of parent
    PCSTR pcszVerb;                // pointer to string with verb names
} URLINVOKECOMMANDINFO;
```

As its first argument, the function expects one of the CSIDL constants that identify the folder. Then the function expects a pointer to a callback function of the `ENUMSHELLCALLBACK` type. This pointer is your interface to the found folders and file objects because `EnumFolders()` passes the found objects through the call of this callback function.

Its prototype is declared in the `ENUMFOLD.H` Include file, which you must add to your programs to get access to the `EnumFolders()` function. Here’s what the declaration looks like:

```
typedef BOOL (WINAPI* ENUMSHELLCALLBACK)(
    LPSHELLFOLDER lpFolder, // Pointer to IShellFolder interface of parent
    PSTR          pszPath,  // Ptr to path and filename (if part of file
                           // system)
    PSTR          pszDisplayName, // Pointer to Displayname of object
    DWORD         ulAttrib,    // SFGAO attributes of object
    DWORD         dwUser,      // specified by caller with EnumFolders()
    LPITEMIDLIST pidlPath,     // absolute Item-Identifier list
    LPITEMIDLIST pidl,         // Item-Identifier of current object
    int           iLevel       // Nesting level
);
```

As its first parameter, the called function gets a pointer to the `IShellFolder` interface of the `ShellFolder` object containing the returned object. If the returned object is an item mapped from the file system to the Shell Namespace, you’ll find a pointer to the complete path and filename of the object as a C string in the `pszPath` parameter. It continues with the display name of the object, to which the `pszDisplayName` parameter refers. You get the different attributes of the object in `ulAttrib`, as you can determine through `IShellFolder::GetAttributesOf`. All the attributes were queried, so you can test the value in `ulAttrib`

with all SFGAO constants. Next, in `dwUser` you get the value that you specified in the original call to `EnumFolders()`. That way a connection between the caller and the program code in the callback function can be established if necessary.

You get two item identifier lists through the `pidlPath` and `pidl`. `pidlPath` parameters. `pidlPath` refers to an item identifier list that receives the complete item ID path from the Desktop to the object. In contrast, `pidl` points to an item identifier list that consists of only the item identifier of the current object (plus the terminating null identifier).

The current nesting level within the Namespace is specified in `iLevel`, which is the last callback parameter. In this context, you need to know that `EnumFolders()` not only browses through the contents of the specified start folder, it also goes into its subfolders to browse the Namespace below the specified start folder completely. The function does so recursively and always returns the current level in `iLevel`. The first call of the callback function is always used to pass the start object, with 0 being specified for `iLevel`.

Within the callback function you can through the function result influence the remainder of the browsing of the Namespace in the framework of `EnumFolders()`. If your callback function returns `TRUE`, `EnumFolders()` continues browsing the Namespace. A return value of `FALSE` causes the function to cancel the recursion branch and not browse any other subfolders in this branch. You'll find examples for using `EnumFolders()` and the concrete coding of the callback function in several programs throughout this chapter.

You'll find the following program(s) on the companion CD-ROM



ENUMFOLD.C (C listing)

The SHORTCUT.C help module

The `SHORTCUT.C` module is used whenever work is done with shortcuts in the sample programs of this chapter. This module contains two powerful functions. First is `CreateShortcut()`. It does the work with the `IShellLink` OLE interface for you and creates a shortcut to any file. You can put the shortcut file anywhere you want in the file system. It could even appear on the Desktop or in one of its folders. The parameters of this function largely correspond to the various attributes of a shortcut, as can be set through the methods of the `IShellLink` interface. You cannot use this function to set the hotkey or the description of a shortcut. It is, however, easy to add these features to the function.

```

BOOL CreateShortcut( LPSTR pszPath,          // Target of shortcut (drive, path,
                                                // filename)
                    LPSTR pszArguments,      // Arguments for the command line
                    LPSTR pszLocation        // Path and filename under which
                                                // the shortcut is saved
                    LPSTR pszWorkingDir,     // Working directory
                    int  nCmdShow );         // Display mode of the application
                                                // window

```

If it was possible to create the shortcut, you get a function result of `TRUE`, otherwise you get `FALSE`.

A look at the listing of the module shows that there aren't any special secrets or tricks to the `CreateShortcut()` function. It does, however, eliminate much of the work for the caller through its variety of operations to be executed. First, OLE access is initialized by `CoInitialize()`, in case this hasn't yet taken place. Next, a new `ShellLink` object is created and the desired properties of the new shortcut are set by the methods of the `IShellLink` interface. Finally, the new shortcut is saved using its `IPersistFile` interface, and OLE access is switched off. That's it.

The module also contains the `SetShortcutIcon()` function, used to set a shortcut's icon. To do this, all you need is the name and path of the shortcut file, the name and path of a DLL or EXE file with the icon, and an index that specifies the number of the icon within the icon file.

You'll find the following program(s) on the companion CD-ROM



SHORTCUT.C (C listing)

44

A Closer Look At The Registry

Traditionally, Windows application and system component settings were stored in separate INI files. The settings are saved from one session to the next through these files. The same function in Windows 95 is handled by a central component called the Registry. All applications claiming to be Windows 95 compatible must refer to the Registry instead of INI files. Windows 95 itself uses the Registry to store settings between sessions. All information will be found exclusively in the Registry. This is true regardless of the type of setting including system component, device driver, desktop configuration or OLE object.

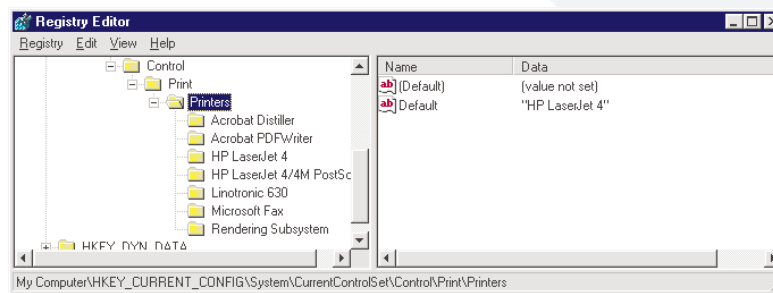
INI files have not been abandoned completely although they're no longer supported by the system. The old Win16 API functions for accessing INI files (GetPrivateProfileString(), WritePrivateProfileString() etc.) are still available for applications to use. The Registry eliminates some of the limitations of INI files. These limitations include their size limit of 64K per file, their inability to store binary data (no strings or integers or their restriction to a two-level hierarchy of sections and entries. Also, several advanced features, such as transparent management of configuration data for multiple users on one machine, are possible only through the central Registry.

To accommodate these new developments, we recommend storing application specific settings in the Registry only. Windows 95 will then support your applications through a special Registry API with functions for all tasks handled by the Registry. These functions are necessary because the Registry isn't treated as an ASCII file but as a binary file that cannot be read or written to in the usual manner. Before talking about the individual Registry API functions, we'll explain the basic idea behind the Registry and its underlying structure.

Registry Organization And Structure

The Registry expands on the old INI structure. In this structure, individual entries within a file were simply grouped into sections. In contrast, as with files, Registry entries can be organized into directories, subdirectories, sub-subdirectories, etc., thus creating a logical grouping. This is the only way to keep track of such a large number of entries.

Structure of the Registry as displayed by REGEDIT



Registry structure

The term "keys" is used when referring to Registry elements instead of directories and subdirectories. Each entry in the Registry represents a key and can contain any number of subkeys. These subkeys, in turn, can have their own subkeys, and so on. In this way a somewhat complex hierarchy is formed, with no limit on the number of levels. Each key can hold one or more key values. Although key values must have names, each key can also have one unnamed key value. This represents the default value for the key.

Viewing the Registry

Use REGEDIT.EXE (included with Windows 95) to view the current structure of the Registry any time. When you first look at the Registry, you might be shocked at the number of keys that exist. The system creates almost 10,000 keys at installation time. Fortunately, you don't need to worry about this large number. What is important is to find the right place for your keys. Use specific defaults to do this. Whatever else that is contained in the Registry does not concern your applications.

The illustration on the previous page shows a key generated by the system called HKEY_CURRENT_CONFIG\System\CurrentControlSet\Controls\Print\Printers. It has eight subkeys. As in path descriptions, the backslashes separate individual keys from their subkeys. The first is HKEY_CURRENT_CONFIG. It contains a subkey called System, among other subkeys. In turn, System, has a subkey CurrentControlSet, which has a subkey Controls, which has Print, which finally has Printers.

The right side of the illustration shows the key values for Printers. One of these is the unnamed default key. It always appears as `_(Default)` in REGEDIT although in reality it has no name at all. In our case there is no value given to this key. In addition you can see that the system has created a key with the name Default. Here is where it stores the name of the current default printer. The system could also generate any number of additional key values. However, here it looks as if no more are needed.

You'll quickly find REGEDIT to be an invaluable tool for programming. Use REGEDIT as a type of debugger when making changes to the Registry. Then check if your program has produced the desired changes.

Naming keys and key values

As with file and directory names, key and key value designations must follow certain rules. The maximum length of a name is 64 characters. Names can contain any ANSI character between 32 and 127 - including spaces. Backslashes and wildcard characters (* and ?) are not allowed. Also, the name cannot begin with a period (such as `_.Software`). All keys beginning with a period are reserved for the system.

Key value types

Unlike with the old INI files, the key value type here is not limited to strings. Besides strings, you can also store integer values as DWORDs or various binary structures. This way an application can group all permanent settings into a data structure and save them under just one key. This is obviously less work than placing each piece of information in a separate key. Now only one key must be loaded when the program starts. Also, you now have the option of hiding the information from unauthorized users. This is done by encrypting the data structure before saving and then decrypting it when loading. The Registry isn't intended as a storage place for entire documents, however. Elements larger than one or two kilobytes should be saved in separate files as before. Configuration data fits easily into the Registry because it normally requires little space.

Separation of user and system data

An important idea with the Registry is the separation of user and system data. While the machine is always the same, users may change from one session to the next. Windows 95 allows each user to set up a personalized workspace with individual settings for colors, Desktop background, Desktop folders and programs, as well as available network resources. A user can log onto any machine in the network and the Desktop will be the same each time. This principle also applies to single PCs shared by many coworkers.

All this is possible because the system uses the Registry to store configuration data for each user individually. When a user logs on, the system chooses the settings based on the user name and password. This is what is known as a `_user profile`.

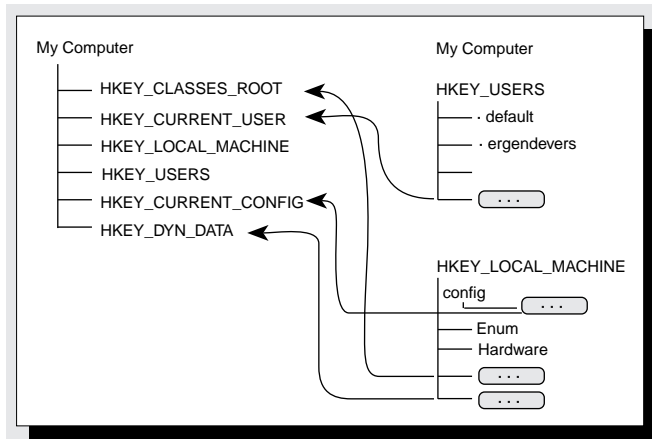
Predefined key structure

The starting point for working through the Registry and creating new keys consists of the keys at the root of the Registry. These are predefined by the system. Six keys are at the root if you view the Registry in REGEDIT. These were set up at the time Windows 95 was installed and cannot be deleted. Also, you cannot add new keys at the root of the Registry. New keys must always go underneath one of the existing keys, which are as follows:

Keys	Meaning
HKEY_USERS	Contains user profiles for the users registered to this computer.
HKEY_LOCAL_MACHINE	Machine configuration, i.e., hardware, installed device drivers and installed software.
HKEY_CURRENT_USER	Contains the user profile for the current user, including program group setup, Desktop appearance, software settings, environment variables etc.
HKEY_CLASSES_ROOT	Used mainly by the Shell and by OLE applications to register OLE classes and OLE documents. OCX controls must be registered here as well.
HKEY_CURRENT_CONFIG	Current configuration settings, such as printer settings and screen fonts.
HKEY_DYN_DATA	Used exclusively by the system and its device drivers for intermediate storage of data.

Unlike the REGEDIT display, the Registry has only two root keys, HKEY_LOCAL_MACHINE and HKEY_USERS. The other four are subkeys of these. HKEY_CURRENT_CONFIG, HKEY_CLASSES_ROOT and HKEY_DYN_DATA are subkeys of HKEY_LOCAL_MACHINE, while HKEY_CURRENT_USER goes under HKEY_USERS. This is invisible on the user level, however.

Various keys are derived from base keys HKEY_LOCAL_MACHINE and HKEY_USERS



Saving application data

The rationale behind HKEY_CURRENT_USER as a subkey of HKEY_USER quickly becomes apparent when you ask yourself where in the Registry the application specific data will be stored. Because of the separation of user specific from machine specific data, there are two answers to this question:

Machine specific data is stored under HKEY_LOCAL_MACHINE, and user specific data under HKEY_CURRENT_USER. Machine specific settings are determined independent of any user by the computer's hardware and resources. Machine specific settings include directories where software is installed, the name of the computer and possibly registration numbers for the software. User specific data includes all settings that a user can enter (through menus and dialog boxes) while a particular application is running. Depending on the type of software, this data could include the user name, the most recent directory for loading documents or a password for protecting data within an application. The system creates a key under HKEY_USERS for each registered user. The key for the currently active user is always under HKEY_CURRENT_USER.

An application loading data from HKEY_CURRENT_USER can sometimes, without realizing it, get the key HKEY_USERS\default (the default user), at other times HKEY_USERS\JulieSpyder and at still other times HKEY_USERS\ScottKiller. In this way, the application can retrieve the settings for the current user without having to know who this user is or where under HKEY_USERS the settings are found.

The applications themselves however should not store anything directly under HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER. The more programs installed the more chaotic the situation would become. Therefore, the first item under both keys will be a subkey called SOFTWARE.

```
HKEY_LOCAL_MACHINE\Software
HKEY_CURRENT_USER\Software
```

A key with the brand name of the software follows the software key, for example, _SuperSoft". We now have:

```
HKEY_LOCAL_MACHINE\Software\SuperSoft
HKEY_CURRENT_USER\Software\SuperSoft
```

Continuing along, we add the name of the actual program. Since all software companies hope to sell more than one product, they should somehow be separated. Here, the product is called _SuperFIBU".

```
HKEY_LOCAL_MACHINE\Software\SuperSoft\SuperFIBU
HKEY_CURRENT_USER\Software\SuperSoft\SuperFIBU
```

Last is the current version number of the program. It's possible for the user to have several versions of the same program on the hard drive. The extra key keeps the settings from becoming mixed up with each other:

```
HKEY_LOCAL_MACHINE\Software\SuperSoft\SuperFIBU\V1.0
HKEY_CURRENT_USER\Software\SuperSoft\SuperFIBU\V1.0
```

Now that the settings can finally be saved, you have several options. First, you could create a subkey for each setting underneath the one already there. Then define the unnamed key values as follows:

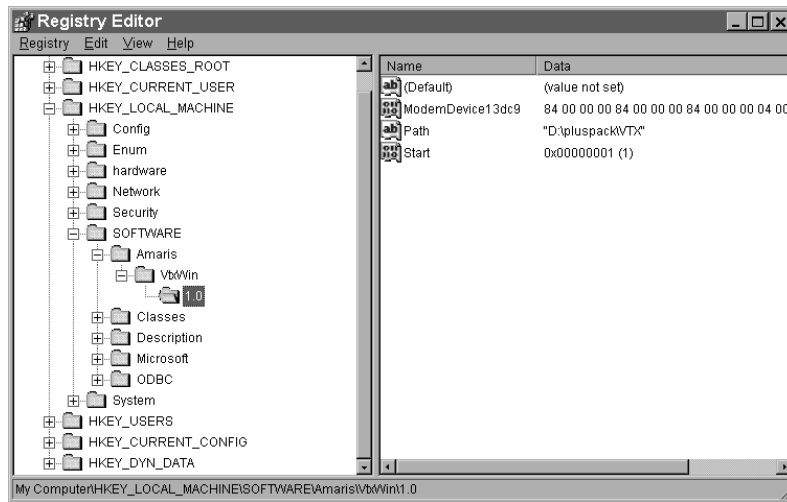
```
HKEY_LOCAL_MACHINE\Software\SuperSoft\SuperFIBU\V1.0\HelpFile = _d:\superFIBU\sف.hlp
HKEY_LOCAL_MACHINE\Software\SuperSoft\SuperFIBU\V1.0\Registration Number = _0x-
12345-abcd"
HKEY_CURRENT_USER\Software\SuperSoft\SuperFIBU\V1.0\WindowBackground = 2
HKEY_CURRENT_USER\Software\SuperSoft\SuperFIBU\V1.0\WindowX = 128
HKEY_CURRENT_USER\Software\SuperSoft\SuperFIBU\V1.0\WindowY = 64
HKEY_CURRENT_USER\Software\SuperSoft\SuperFIBU\V1.0\LastFile = _c:\letters\Grandma
turns 80.ggk"
etc.
```

An easier way, however, is saving the various settings as distinct key values of the version key. This is shown in the following illustration. Here you can see the machine specific settings as they are stored in the Registry for the AMARIS BTX Decoder. AMARIS BTX Decoder is part of the Microsoft PlusPack for Windows 95. This method saves storage space and speeds access to the Registry. Multiple subkeys occupy more space and demand more overhead than multiple key values for the same subkey.

Remember, the key hierarchy described here is not mandatory. By using the appropriate code, it's your application that generates the desired keys, enters the settings and reads them in again the next time the program starts. The system has no influence on this whatsoever. You can follow the pattern above or create a key directly under HKEY_CURRENT_USER with the name of the application. Then store all its settings there.

However since most programs follow the above-mentioned scheme, we recommend that you also do the same. This way if you need to manually change settings later it will be easier to find your way.

*In the overview
you'll see
several key
values*



Basic program call

Besides creating its Registry settings, an application can also direct the system through various predefined keys. Let's take the example of starting a program with the Run command from the Start menu in the Task Bar. Normally, the user must enter the program name (the name of the EXE file) along with the path where the EXE file can be found. You can spare the user this extra work by saving the program's path in the Registry. Now the only thing that needs to be entered is the program name. The system automatically gets the path from the Registry.

The above information does not come from just any key. You must save it in a special key that goes under

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\App Paths
```

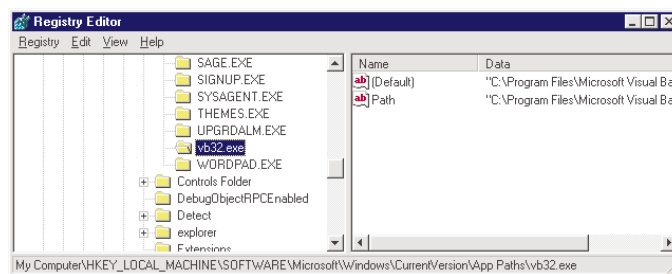
This key must have the same name as the corresponding program file, for example SuperFIBU.EXE. The unnamed key value is the command line call, including the path, along with any required switches.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\App  
Paths\SuperFIBU.EXE = _c:\superfibu\superfibu.exe /d"
```

Besides the unnamed key value, you can also specify another key value with the name `_Path`". The system will use this value as the working directory for the application. So, if the application uses DLLs or other files, simply set the key value `Path`. You can now load files within the application without entering an explicit path.

To save work for the user, create these keys immediately when installing the application. The following illustration shows settings for the 32-bit version of Visual Basic.

*Settings for the
32-bit version of
Visual Basic*

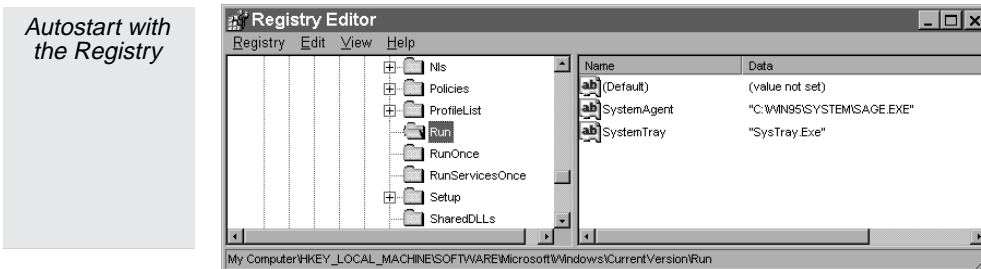


Autostart

The Registry is also responsible for automatic startup of applications at system start. The information is stored under the key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

The RUN key is already provided. Simply add a key value for each program you wish to start automatically. You can choose any names you like if they are distinct from the other names listed. The key values here are command line program calls, for example:

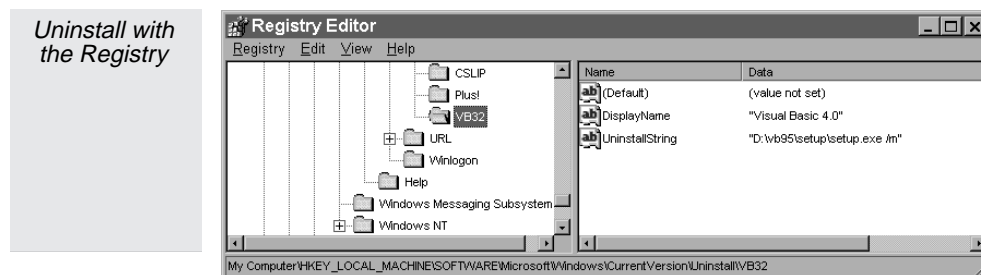


Uninstall preparations

Removing installed applications can also be simplified by creating an appropriate key. Windows 95 has a special program in the Control Panel for adding and deleting programs. If an application has an uninstall program, it should inform the Control Panel by creating a key.

As the illustration below shows, the key must be stored under

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall



The key's name must be the program name. Also, it must consist of two named key values: DisplayName and UninstallString. The value of DisplayName is the complete program name as it should appear in the Control Panel. UninstallString is the command for calling the application's uninstall program, including the path and any required command line switches.

Registry elements

The separation of current machine settings from user settings is also reflected in the physical structure of the Registry. The Registry consists of not one but two separate files called USER.DAT and SYSTEM.DAT. USER.DAT contains all user specific settings (HKEY_USER), while SYSTEM.DAT contains the machine specific settings (HKEY_CURRENT_MACHINE).

What's special here is that the USER.DAT file does not necessarily have to reside on the local machine. It can be accessed within a network through a common server. This way users can switch from one machine to another on the network without reconfiguring their settings each time. The settings are already known thanks to the common USER.DAT file.

Losing the Registry is usually more disastrous than losing a few INI files under Windows 3.1. So, each time Windows 95 starts, it automatically creates a backup copy of the Registry database. It creates a file called USER.DA0 for USER.DAT. Similarly a file called SYSTEM.DA0 is created for SYSTEM.DAT. Upon partial or complete destruction of the Registry, Windows 95 can use the backup copy to restore the lost entries (you'll then need to restart the system).

Accessing The Registry Through Win32 API

You can now see that applications require a wide range of functions before they can use the Registry. They must be able to create new keys and delete them, set key values and retrieve them. Applications also must do other functions as well. The Win32 API provides all these functions through the Registry API.

Win32 functions for accessing the Registry		
Function	Task	
RegOpenKey	Opens a desired key.	Old function
RegOpenKeyEx	Expanded version of RegOpenKey().	New function.
RegQueryValue	Returns the value associated with a key	Old function.
RegQueryValueEx	Returns the value associated with a key.	New function.
RegSetValue	Sets the value associated with a key.	Old function.
RegSetValueEx	Sets the value associated with a key. .	New function
RegFlushKey	Writes changed keys and key values directly to the Registry file.	
RegEnumKey	Enumerates the subkeys for a given key.	
RegEnumKeyEx	Expanded version of RegEnumKey().	
RegEnumValue	Enumerates the values of a given key.	
RegQueryInfoKey	Returns information about a particular key.	
RegCloseKey	Closes a previously opened key.	
RegCreateKey	Creates a new key.	Old function.
RegCreateKeyEx	Expanded version of RegCreateKey().	New function.
RegDeleteKey	Deletes a key and its subordinate entries from the Registry.	
RegDeleteValue	Deletes the value associated with a key.	
RegLoadKey	Loads a key from a file and adds it to the Registry.	
RegSaveKey	Saves a key and all subordinate keys to a file.	
RegReplaceKey	Replaces a section of the Registry by one stored in a file.	
RegUnloadKey	Unloads a section previously loaded by RegLoadKey().	

Many of these functions were already available under Windows 3.1 in the era of the 16-bit API. Windows 3.1, however, only used the Registry to manage OLE classes. Therefore, the functions had to be upgraded slightly for the Win32 API. For example, the Win3.1 Registry recognized only strings as key values and no DWORDs or binary structures. Also each key had only one unnamed key value and could not accept additional named ones.

To preserve compatibility with existing software, the old functions remained unchanged and to them were added the expanded Ex functions. For example RegQueryValueEx() corresponds to RegQueryValue(). If you're not using existing program code, always use the new functions under Win32 (as indicated in the descriptions).

Including Registry functions

Registry API functions are defined in the Include file WINREG.H. Here also is a definition for two basic types that you'll see repeatedly when calling these functions: HKEY and PHKEY. Use key handle HKEY to reference keys from the Registry. Like all Win32 handles, it consists of a 32-bit value. Its complement is a pointer type called PHKEY. It points to a variable

of type HKEY. Many functions that return a key handle require a parameter of type PHKEY pointing to an HKEY variable. This variable contains the desired key handle after the function runs.

```
DECLARE_HANDLE ( HKEY );
typedef HKEY *PHKEY;
```

Access to Registry keys always begins at the root, with one of the predefined keys such as HKEY_CURRENT_USER or HKEY_LOCAL_MACHINE. The corresponding constants are therefore defined in WINREG.H.

```
#define HKEY_CLASSES_ROOT          (( HKEY ) 0x80000000 )
#define HKEY_CURRENT_USER         (( HKEY ) 0x80000001 )
#define HKEY_LOCAL_MACHINE        (( HKEY ) 0x80000002 )
#define HKEY_USERS                 (( HKEY ) 0x80000003 )
#define HKEY_CURRENT_CONFIG       (( HKEY ) 0x80000005 )
#define HKEY_DYN_DATA              (( HKEY ) 0x80000006 )
```

Unlike other APIs, Registry API functions continuously return long values that indicate the status of each function call. If the function was executed successfully, you get ERROR_SUCCESS; otherwise, an error code results. In this case, the API function GetLastError() provides information about the error.

A whole series of functions exists for what are very simple tasks. Fortunately, the ideas and parameters for these functions are very similar. Therefore, they won't take long to learn.

Differences between Windows 95 and Windows NT

Although the Registries of Windows 95 and Windows NT are based on the same API functions, a few small but distinct differences exist. Some of these functions require security descriptors for entering or validating access rights to keys. Since Windows/95 doesn't use the Windows NT security system, these particular parameters have no meaning under Windows/95. Simply enter a value of NULL for them.

The same is true for certain functions that accept or return a class name. Class names are associated with OLEs and serve to link a key with its owner, or generator. This feature also remains unused under Windows 95. The respective parameters can be assigned a value of 0 or NULL, depending on whether they are integers or pointers.

Accessing keys and their values

To determine the value of a key or whether a certain key exists, begin at the top of the Registry. Then work your way down the Registry. The functions RegOpenKey() and RegOpenKeyEx() take you one or more levels deeper into the hierarchy. You always have to start from a key that's already open (whose key handle has been made available to you).

The calls to RegOpenKey() and RegOpenKeyEx() are identical except that two extra parameters exist for RegOpenKeyEx(). One of these is reserved and has no function (at least in this version of Windows 95):

```
LONG RegOpenKey ( // Win16
    HKEY    hKey, // Handle of desired key
    LPCSTR  lpSubKey, // Pointer to C-string with name of subordinate key
    PHKEY   phkResult // Pointer to HKEY variable containing key handle
);
LONG RegOpenKeyEx ( // Win32
    HKEY    hKey, // Handle of desired key
    LPCSTR  lpSubKey, // Pointer to C-string with name of subordinate key
    DWORD   ulOptions, // Reserved, 0
    REGSAM  samDesired, // Access rights desired for key handle
    PHKEY   phkResult // Pointer to HKEY variable containing key handle
);
```


The only important difference is in the `SamDesired` parameter for `RegOpenKeyEx()`. It determines what operations can be performed on the key handle returned. Before we go into this however, let's start at the beginning.

The first parameter for both functions is the key handle for the key above the one being opened. At the highest level of the Registry, the first call to the function must specify one of the predefined constants `HKEY_LOCAL_MACHINE`, `HKEY_CLASSES_ROOT`, `HKEY_LOCAL_MACHINE` or `HKEY_USERS`. The lower level key to be opened is given to the API by the parameter `lpSubKey`, a pointer to a character string containing the name of the subordinate key. The name can be a single key (for example `_Software`) or it can be a complete path such as `Software\Microsoft\Windows\CurrentVersion\App Paths`. As with all functions that require key names, upper case and lower case are ignored.

If the desired key is found, its key handle is saved in the variable pointed to by `phkResult`. This must, of course, be a variable of type `HKEY`. The result `ERROR_SUCCESS` tells you that the key was found and the variable properly loaded. Otherwise, you know something went wrong and the key could not be found.

The returned key (its key handle) can now be used to make further calls to `RegOpenKey()` or `RegOpenKeyEx()` thereby opening further subkeys. To access a key like `_Software\Microsoft\Windows\CurrentVersion\App Paths` call `RegOpenKey()` or `RegOpenKeyEx()` five times in succession. Then continue from key to key. You would specify a single subkey with each call (first `_Software`, then `_Microsoft`, then `_Windows` etc.), each time referring to the key handle obtained by the previous function call. Of course, you could also save yourself these steps by entering the complete path all at once.

With `RegOpenKey()` all `Reg...` functions can be used on the key handle returned. `RegOpenKeyEx()` however allows you to explicitly state the uses for this handle through the parameter `SamDesired`. The `Sam` stands for `_security admission`, another term for access rights. The `SamDesired` parameter can consist of the following constants or combinations thereof:

Constants	Meaning
<code>KEY_ALL_ACCESS</code>	All access rights as described below
<code>KEY_CREATE_SUB_KEY</code>	Allows creation of subordinate keys
<code>KEY_ENUMERATE_SUB_KEYS</code>	Allows access to subordinate keys
<code>KEY_QUERY_VALUE</code>	Gives read access for key values
<code>KEY_READ</code>	Combines <code>KEY_QUERY_VALUES</code> , <code>KEY_ENUMERATE_SUB_KEYS</code> and <code>KEY_NOTIFY</code>
<code>KEY_SET_VALUE</code>	Allows modification of key value
<code>KEY_WRITE</code>	Combines <code>KEY_SET_VALUE</code> and <code>KEY_CREATE_SUB_KEY</code>

`RegOpenKeyEx()` also allows you to obtain a key handle for a key whose handle you already have. For example, say you are calling a subprocess and wish to pass a key handle to it, but this key handle shouldn't have all the access capabilities as the original. In this case `HKEY` would be the key to be passed and the `lpSubKey` pointer would be `NULL`. This tells the function not to look for a subkey, but to use the key given in `HKEY`.

The following code uses this method to open the key for automatic startup of applications at the start of Windows 95.

```
#define AUTORUN_KEY      HKEY_LOCAL_MACHINE
#define AUTORUN_SUBKEY   "Software\\Microsoft\\Windows\\CurrentVersion\\Run"

HKEY hAutoRun;

// Open Autorun key
if( RegOpenKeyEx( AUTORUN_KEY, // Starting point
                AUTORUN_SUBKEY, // Key path
                0, // Reserved
                KEY_ALL_ACCESS, // All access rights
                &hAutoRun ) == ERROR_SUCCESS ) // Pointer to key handle var.
    printf( "_Key-Handle: 0x%x\n", hAutoRun );
```

Returning key handles

As with other handles, key handles should be returned to the system when no longer needed. Return of handles can occur at the end of a program or even as the Registry hierarchy is being traversed. When a subordinate key is opened by `RegOpenKeyEx()` you can release the higher level key handle if you won't be referencing it in subsequent `Reg...` functions. Once you have the handle for a desired key, the system no longer cares whether you also still have the higher level key handles. Only the desired key handle is what counts.

Open key handles are returned to the system through the function `RegCloseKey()`. Its sole argument is the key handle to be returned. The only exceptions here are the predefined key handles such as `HKEY_CURRENT_USER` etc., which must never be closed.

```
LONG RegCloseKey (
    HKEY hKey    // Key handle to be returned
);
```

When used on a valid open key handle, the function returns the result `ERROR_SUCCESS`. This handle can no longer be used in subsequent calls to `Reg...` functions.

Reading key values

Once you have found your way to a certain key in the Registry hierarchy, you'll usually want to know its value. For this purpose we have the function `RegQueryValueEx()`, which requires a key handle as a parameter.

```
LONG RegQueryValueEx (
    HKEY hKey, // Handle of desired key
    LPCSTR lpValueName, // Pointer to C-string with name of key value
    LPDWORD lpReserved, // Reserved, NULL
    LPDWORD lpType,      // Pointer to variable containing type information
    LPBYTE lpData,       // Pointer to buffer containing value
    LPDWORD lpcbData     // Pointer to variable containing size of value buffer
);
```

The second parameter is a pointer to a character string and must contain the name of the desired key value. Since a key can contain many key values you must differentiate between entries. Besides multiple named key values a key can also have one unnamed key value. To determine its value, simply enter `NULL` for `lpValueName`.

The `lpType` parameter receives the address of a `DWORD`. Upon successful function execution, this `DWORD` contains the type of key value returned. It then becomes one of the flags listed in the table below. The location of the key value is given by the `lpData` parameter. This is a pointer to a buffer provided by the caller. To keep the buffer from overflowing, the last parameter `lpcbData` must specify a pointer to a `DWORD`. The calling program must load the buffer size into this `DWORD` before calling `RegQueryValueEx()`. If the function is successful the `DWORD` will contain the number of bytes used for storing the key value. If you then want to save the key value in another variable, only this portion of the buffer needs to be copied.

A successful read of the desired key value is signaled by the result `ERROR_SUCCESS`. If the key value was found but not enough space for it existed in the buffer, you get `ERROR_MORE_DATA`. In this case, increase the buffer size based on the value returned in the variable pointed to by `lpcbData`. Then rerun the function.

When you're unsure of the size of the key value and expect the buffer size to be too small, use the following simple trick. Enter any value greater than 0 into the variable containing the buffer size. Enter a value of `NULL` for the `lpData` parameter. The function then knows you're looking for the key value. It will give back its length in bytes but will not query the actual value. It returns `ERROR_SUCCESS` without even trying to copy the key value into a buffer.

So, first call `RegQueryValueEx()` to determine the required buffer size and allocate the buffer through the local heap. Then use the pointer returned by `malloc()` to call `RegQueryValueEx()` again.

The following code sequence illustrates this process using the Help text of the Explorer as an example. It's stored under the key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Tips`. Individual tips are stored as key values, with sequential numbers as key value names. The program queries the numbers from 0 to 99 and outputs their values if they exist. For each tip the first call to `RegQueryValueEx()` gives the size of the tip. Memory is then allocated through the heap. Finally, the tip is read in by a second call to `RegQueryValueEx()`.

```
#include <windows.h>
#include <stdio.h>

#define TIP_KEY      HKEY_LOCAL_MACHINE
#define TIP_SUBKEY   "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Tips"

void main( void )
{
    HKEY hTipKey;

    if( RegOpenKeyEx( TIP_KEY,
                     TIP_SUBKEY,
                     0,
                     KEY_ALL_ACCESS,
                     &hTipKey ) == ERROR_SUCCESS )
    {
        char chValueName[ 32 ];
        int i;
        for( i = 0; i < 99; i++ )
        {
            DWORD dwType, cbData;
            wsprintf( chValueName, "%ld", i );

            if( RegQueryValueEx( hTipKey,
                                chValueName,
                                NULL,
                                &dwType,
                                NULL, // Don't read data yet
                                &cbData ) == ERROR_SUCCESS )
            {
                LPBYTE lpData;
                lpData = malloc( cbData + 1 );
                if( lpData )
                {
                    if( RegQueryValueEx( hTipKey,
                                        chValueName,
                                        NULL,
                                        &dwType,
                                        lpData, // Don't read data yet
                                        &cbData ) == ERROR_SUCCESS )
                    {
                        printf( "Tip %s: %s\n", chValueName, lpData );
                        free( lpData );
                    }
                }
            }
        }
        RegCloseKey( hTipKey );
    }
}
```

```
}
```

The contents of the buffer with the returned key value is described by the DWORD variable whose address was given in the lpType parameter. This variable contains one of the following constants following successful execution:

Flag	Data type
REG_BINARY	An undefined, binary structure
REG_DWORD	A DWORD (32-bit)
REG_DWORD_LITTLE_ENDIAN	A DWORD in Little Endian format
REG_DWORD_BIG_ENDIAN	A DWORD in Big Endian format
REG_SZ	A C-string
REG_MULTI_SZ	An array of C-strings
REG_EXPAND_SZ	A C-string with expanded environment variables
REG_NONE	No value exists

The following paragraphs briefly discuss the individual formats. With REG_BINARY you need to know the data format to interpret its content. If you've stored the data there yourself this shouldn't be a problem.

The DWORD types REG_DWORD and REG_DWORD_LITTLE_ENDIAN are identical because Intel processors (the basis of the Windows 95 operating system) store all data in memory in Little Endian format. In Little Endian format, the Lo word of the DWORD is stored first at address N in memory. Then the Hi word is stored at address N+2. The same happens with the Hi and Lo bytes of each word. The Lo byte always goes before the Hi byte in memory.

Exactly the opposite is true for data type REG_DWORD_BIG_ENDIAN. The words and bytes in the DWORD are switched around. This is the format used by Motorola processors as well as many other RISC processors. However, you'll never see it under Windows 95. You'll only see it under Windows NT if you're using a non-Intel machine.

REG_SZ, REG_MULTI_SZ and REG_EXPAND_SZ refer to C-strings ending with a NULL byte. While REG_SZ and REG_EXPAND_SZ each consists of one string, REG_MULTI_SZ is an entire array with one string immediately following another. The end of the array is marked by an empty string (a NULL byte).

REG_SZ and REG_EXPAND_SZ differ in only one (and trivial in many cases) aspect. It concerns expansion of environment variable placeholders within the string. For example, in a REG_EXPAND_SZ string the familiar %PATH% is automatically replaced by the current setting for this environment variable. So, if the key value contains _Hello;%PATH%;Hello" you would get _Hello;C:\DOS;C:\WINDOWS95....;Hello" in return. REG_SZ keeps the environment variable placeholder the same so your return value would be _Hello;%PATH%;Hello".

Besides RegQueryValueEx(), Win32 continues to support the standard version of the function, RegQueryValue(). The old version however should be avoided, in favor of RegQueryValueEx().

```
LONG RegQueryValue (          // Old 16-bit version, avoid under Win32!
    HKEY    hKey, // Handle of desired key
    LPCSTR  lpSubKey, // Pointer to C-string with name of key value
    LPSTR   lpValue, // Pointer to buffer containing key value
    PLONG   lpcbValue // Pointer to variable containing length of key value
);
```

Setting the key value

The counterparts to RegQueryValue() and RegQueryValueEx() respectively are the functions RegSetValue() and RegSetValueEx(). Use the expanded Ex function instead of the standard one established under Win16. The main difference regarding parameters and options between these functions is that RegSetValue() accepts only C-strings as the key value while RegSetValueEx() accepts all Registry entry types.

```

LONG RegSetValue ( // Win16
    HKEY    hKey, // Handle of desired key
    LPCSTR  lpValueName, // Pointer to C-string with name of key value
    DWORD   dwType,      // Type of key value, always REG_SZ
    LPCSTR  lpData,      // Pointer to buffer with C-string as key value
    DWORD   cbData // Buffer size in bytes
);
LONG RegSetValueEx ( // Win32
    HKEY    hKey, // Handle of desired key
    LPCSTR  lpValueName, // Pointer to C-string with name of key value
    DWORD   Reserved,    // Reserved, 0
    DWORD   dwType,      // Type of key value (REG_BINARY etc.)
    LPCBYTE lpData,      // Pointer to buffer with key value
    DWORD   cbData       // Buffer size in bytes
);

```

Besides the key handle, these functions require a pointer to a C-string in `lpValueName` containing the name of the key value being set. Enter NULL instead of an actual pointer for an unnamed key value. The key value type goes in `dwType`. While `RegSetValue()` allows only `REG_SZ`, `RegSetValueEx()` accepts all `REG_...` constants as we discussed under `RegQueryValueEx()`. For example, you could use `REG_DWORD` or `REG_BINARY`, in which case the pointer in `lpData` would point to a `DWORD` or the beginning of a data structure in memory. The size of this element (in bytes) is given by the `cbData` parameter. If the desired value was successfully set, the function returns `ERROR_SUCCESS`. Otherwise, an error code is given.

The following example uses the `RegSetValueEx()` function to set two Windows system keys, the name of the licensed user and the user's company. These values are found under the keys `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RegisteredOwner` and `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RegisteredOrganization` and can easily be changed. You can see the changes immediately in the Help window of the Explorer by calling up `_Help About Windows 95`.

```

#include <windows.h>

#define LICENSE_KEY    HKEY_LOCAL_MACHINE
#define LICENSE_SUBKEY "Software\\Microsoft\\Windows\\CurrentVersion"

void main( void )
{
    HKEY hLicenseKey;

    if( RegOpenKeyEx( LICENSE_KEY,
                     LICENSE_SUBKEY,
                     0,
                     KEY_ALL_ACCESS,
                     &hLicenseKey ) == ERROR_SUCCESS )
    {
        #define NEW_OWNER "MITI & BHJ"
        #define NEW_ORG   "Software for All!"

        RegSetValueEx( hLicenseKey,
                       "RegisteredOwner",
                       0,
                       REG_SZ,
                       NEW_OWNER,

```

```

        lstrlen( NEW_OWNER ) );

    RegSetValueEx( hLicenseKey,
        "RegisteredOrganization",
        0,
        REG_SZ,
        NEW_ORG,
        lstrlen( NEW_ORG ) );

    RegCloseKey( hLicenseKey );
}
}

```

Saving a changed key

When you change a key or its key values, the changes don't immediately go to the Registry file on disk. Instead the system establishes a thread that is responsible for flushing the changed keys in the background. The thread has relatively low priority so it's used when nothing else is happening in the system. This is usually acceptable. However, you run the risk of losing your changes if the system crashes because they haven't yet been saved.

So, if you suspect the system won't get around to flushing the changed keys, initiate this process explicitly yourself. Use the function `RegFlushKey()`. Remember to use it only in emergency situations.

```

LONG RegFlushKey (
    HKEY hKey    // Handle of key to be saved
);

```

Searching through all subordinate keys

While functions such as `RegOpenKey()`, `RegQueryValueEx()` and `RegSetValueEx()` always access a specific (and therefore known) key, other `Reg...` functions allow you to search through all subordinate keys and their key values. When searching for a particular key or Registry without knowing its key structure, use functions `RegEnumKey()` and `RegEnumKeyEx()` to search through subordinate keys and `RegEnumValue()` to search through key values.

Let's first look at `RegEnumKey()` and `RegEnumKeyEx()`. Again we have the old Win16 function along with its new Win32 counterpart. `RegEnumKeyEx()` is clearly preferable. While `RegEnumKey()` gives you the name of the subordinate key, `RegEnumKeyEx()` also gives you the key's class name and the date and time it was created and last modified.

```

LONG RegEnumKey ( // Win16
    HKEY hKey,    // Handle of desired key
    DWORD dwIndex, // Index of key value to be queried
    LPSTR lpName, // Pointer to buffer receiving name
    DWORD cbName  // Buffer size in characters
);

LONG RegEnumKeyEx ( // Win32
    HKEY hKey,    // Handle of desired key
    DWORD dwIndex, // // Index of key value to be queried
    LPSTR lpName,  // Pointer to buffer receiving name
    LPDWORD lpcbName, // Pointer to variable containing size of name buffer
    LPDWORD lpReserved, // Reserved, NULL
    LPSTR lpClass,  // Win95: NULL
    LPDWORD lpcbClass, // Win95: NULL
    PFILETIME lpftLastWriteTime // Pointer to FILETIME var, which contains the date
);

```

The first parameter, as usual, is the handle of the key whose subordinate keys we want to find. When starting at the beginning of the Registry, enter one of the known constants `HKEY_USERS`, etc. The next parameter is the number (index) of the subordinate key for whose name we are searching. The trick to using these functions is to enter 0 for the first call, 1 for the second, 2 for the third, etc. This way you get the name of each subordinate key in succession.

The name is returned into a buffer that you must allocate for this purpose. Then use the parameter `lpName` to pass it to the function. `cbName` is a pointer to a variable containing the maximum number of characters in the buffer. If the function call is successful this variable will contain the length of the key name stored in `*lpName`.

Another buffer pointer/size pair is `lpClass` and `lpcbClass`. This buffer is where the function writes the key's class name. Since class names are not supported under Windows 95, simply enter a value of `NULL`.

The function result tells you if a further subordinate key could be found. When the result equals the constant `ERROR_SUCCESS`, the function was successful and has filled the passed buffer with a character string containing the key's name.

At this point we suggest incrementing the index and calling the function again. If no additional keys exist under this index you get a result of `ERROR_NO_MORE_ITEMS`. This result means that all subordinate keys were searched (assuming you called the function for all prior index values).

The following code sequence shows how `RegEnumKeyEx()` is used within the framework of the recursive function `RegEnum()`. All subordinate keys of a given key are searched. Their names are output to the screen. The names are indented according to their level within the key hierarchy. The `main()` function uses `RegEnum()` to output the names of all keys below `HKEY_CURRENT_USER`.

```
#include <windows.h>
#include <stdio.h>

void RegEnum( HKEY hKey, int iLevel )
{
    int iSubKey;
    char chSubKey[ 256 ];
    DWORD cbSubKey;
    FILETIME ft;

    iSubKey = 0;
    cbSubKey = sizeof( chSubKey );

    //List all keys until an error occurs
    while( RegEnumKeyEx( hKey,
                        iSubKey,
                        chSubKey,
                        &cbSubKey,
                        NULL,
                        NULL,
                        NULL,
                        &ft ) == ERROR_SUCCESS )
    {
        int j;
        HKEY hSubKey;

        // Output key names
        for( j = 0; j < iLevel; j++ ) printf( "  ");      // Indent
        printf( "%s\n", chSubKey );
    }
}
```

```

// Open subkey
if( RegOpenKeyEx( hKey,
                  chSubKey,
                  0,
                  KEY_ALL_ACCESS,
                  &hSubKey ) == ERROR_SUCCESS )
{
    RegEnum( hSubKey, iLevel + 1 ); // Recursion
    RegCloseKey( hSubKey );        // Housekeeping
}
iSubKey++; // Enumerate next key
cbSubKey = sizeof( chSubKey );
}
}

void main( void )
{
    RegEnum( HKEY_CURRENT_USER, 0 );
}

```

Searching through key values

Similarly, you can also search the key values of a key by passing its key handle to the function `RegEnumValue()`. Unlike the others, `RegEnumValue()` has no corresponding `Ex` function; it was introduced only with Win32. As above, the function is called repeatedly to sequentially evaluate each key value and its name. Start with a `dwIndex` parameter of 0, followed by 1, 2, 3 etc. until the function returns `ERROR_NO_MORE_ITEMS` instead of `ERROR_SUCCESS`.

```

LONG RegEnumValue (
    HKEY      hKey, // Handle of desired key
    DWORD     dwIndex, // Number of key value to be read
    LPSTR     lpValueName, // Pointer to buffer receiving key value name
    LPDWORD   lpcbValueName, // Pointer to variable receiving key value length
    LPDWORD   lpReserved, // Reserved, NULL
    LPDWORD   lpType, // Pointer to variable receiving key value type
    LPBYTE     lpData, // Pointer to buffer for key value
    LPDWORD   lpcbData // Pointer to variable with length of key value buffer
);

```

The parameters `lpValueName` and `lpcbValueName` follow the index. They refer to the calling program's buffer. When the function finds the name of a key value, it stores it here as a C-string. You must specify the buffer length in `lpcbValueName` before calling the function. Then the function can check if there's enough space for the name that it finds. If not, the function returns empty to the caller along with the appropriate error code. On the other hand, if the function is successful, `lpcbValueName` will contain the actual number of bytes used for the name.

If this number equals 1, only the first byte of the buffer was used. Therefore, an empty string that consists of one NULL byte was returned. This signifies an unnamed key value.

Like `lpcbValueName` the `lpType` parameter is also a pointer to a `DWORD`. The caller must initialize this parameter with the address of the int variable that will contain the type of key value returned. The `RegEnumValue()` function loads this variable with one of the `REG...` constants previously discussed, e.g. `REG_DWORD` or `REG_SZ`.

The storage location for the key value is determined by the parameters `lpData` and `lpcbData`. Following the usual pattern, `lpData` is a pointer to the (previously allocated) buffer. `lpcbData` is a pointer to a `DWORD` variable containing the length of this buffer in bytes. If all goes well, the variable will contain the number of bytes used to store the key value in the buffer.

Not one byte too many

With functions such as `RegEnumValue()` and `RegEnumKey()`, allocating buffers up front can be a major inconvenience. If you allocate too little space the functions return an error. If you decide to play it safe instead, you might get only a few bytes of `REG_BINARY` data when you were expecting 4K.

Some assistance is available through the `RegQueryInfoKey()` function. It gives information about a key and the key directly beneath it. One item this function returns is the maximum length of a key name or key value. So, before searching through subordinate keys, run `RegEnumKeyEx()` or `RegEnumValue()` to determine how many bytes are needed for the buffer.

```
LONG RegQueryInfoKey (
    HKEY    hKey,        // Handle of desired key
    LPSTR    lpClass,     // Windows 95: NULL

    // The following parameters are pointers to DWORD variables (int).
    // Upon successful function execution they contain the following
    // information:

    LPDWORD   lpcbClass, // Windows 95: NULL
    LPDWORD   lpReserved, // Reserved, NULL
    LPDWORD   lpcSubKeys, // Number of subordinate keys
    LPDWORD   lpcbMaxSubKeyLen, // Maximum length of a subordinate key name
    LPDWORD   lpcbMaxClassLen, // Maximum length of a class name
    LPDWORD   lpcValues, // Number of key values for HKEY
    LPDWORD   lpcbMaxValueNameLen, // Maximum length of a key value name
    LPDWORD   lpcbMaxValueLen, // Maximum length of a key value
    LPDWORD   lpcbSecurityDescriptor, // Win95: 0
    PFILETIME lpftLastWriteTime // Pointer to FILETIME var with date of last
modification
);
```

In the commented prototype above, notice the function returns most of the information in unsigned int variables (DWORDs). The addresses of these variables are given by the caller as function parameters.

The starting point for this function is, as always, the key handle of a previously opened key. First, the function returns information about the key such as class name and length, the number of key values and the maximum length of a key value or name. This information simplifies subsequent calls to `RegEnumValue()` that refer to this key.

The function also returns information about the subkeys of the given key. This specifically includes the number of subkeys as well as the maximum length of a key name and its associated class name. In each case the values returned represent the actual length of the string and do not include the final NULL byte that comes with a C-string. So, when allocating a buffer based on the maximum length returned (N), enter a length of N+1.

Creating new keys

The `Reg...` functions we've looked at so far apply to already existing keys. However, it's also possible for applications to create new keys at runtime, and then create subkeys, assign key values, etc. This process is initiated by the two functions `RegCreateKey()` and `RegCreateKeyEx()`, which allow the creation of new keys underneath an already open key. We recommend not using `RegCreateKey()` under Win32 because it's based on the Win 3.1 version of the Registry. Use `RegCreateKeyEx()` instead.

```
LONG RegCreateKey (           // Win16
    HKEY    hKey, // Handle of key above new key
    LPCSTR  lpSubKey, // Pointer to C-string with name of new key
    PHKEY   phkResult // Pointer to variable receiving key handle of new key
```

```

);
LONG RegCreateKeyEx (      // Win32
    HKEY    hKey, // Handle of higher level key
    LPCSTR  lpSubKey, // Pointer to C-string with name of key being generated
    DWORD   Reserved, // Reserved, 0
    LPSTR    lpClass, // Win95: NULL
    DWORD    dwOptions, // Win95: REG_OPTION_NON_VOLATILE
    REGSAM   samDesired, // Access rights (KEY_ALL_ACCESS etc.)
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // Win95: NULL
    PHKEY    phkResult, // Pointer to variable receiving key handle
    LPDWORD  lpdwDisposition // Pointer to variable receiving status
);

```

Creating a new key must always start with a key already opened by `RegOpenKey()` or `RegOpenKeyEx()` or one of the predefined keys at the root of the Registry (`HKEY_CURRENT_USER` etc.). The only exception is when no new keys can be created under `HKEY_USERS` - this is reserved for the system.

The first parameter of `RegCreateKeyEx()` is the key handle of the higher level key, followed by `lpSubKey`, a pointer to a C-string containing the name of the key being created. Here you can have a single key (`_Software`) or a complete path name such as `_Software\Microsoft\Windows\CurrentVersion\App Paths`, in which case the function generates all keys along the given path. It doesn't matter if part of the path or even the entire path already exists. The function simply expands the existing path accordingly.

The next parameter is reserved and must equal 0. Next is another pointer to a C-string which determines the class name of the new key. As always in Windows 95, enter `NULL` for the class name.

The `dwOptions` parameter applies only to Windows NT and should be set to 0 in Windows 95. `samDesired` determines the level of access the caller will have to the new key. As described under `RegOpenKeyEx()`, this involves some combination of the flags `KEY_ALL_ACCESS`, `KEY_READ`, `KEY_WRITE` etc. The simplest is entering `KEY_ALL_ACCESS`, which allows the returned key handle to be used by all `Reg...` functions.

The next parameter is a security descriptor. It's also applicable only to Windows NT. Its value is `NULL` under Windows 95.

`phkResult` determines where the key handle for the new key is stored. It gives the address of an `HKEY` variable. Similarly for the last parameter `lpdwDisposition` - the caller must initialize this parameter with the address of an unsigned int variable (`DWORD`). The function then places into this variable one of the following two constants, indicating whether the desired key was created or it already existed.

<code>REG_CREATED_NEW_KEY</code>	The desired key did not exist and was created
<code>REG_OPENED_EXISTING_KEY</code>	The desired key already existed and was opened

With `REG_OPENED_EXISTING_KEY` the desired key was found to already exist and was opened the same as with a call to `RegOpenKeyEx()`. Although the key wasn't generated, you can still access it normally through the key handle returned.

Before inspecting the contents of the variable referenced by `lpdwDisposition`, take a look at the function result. If it's not `ERROR_SUCCESS` it means an error has occurred and you don't need to check the other variables.

The following code shows how to generate keys and key values so you can call `MYPROG.EXE` via Start/Run.

```

#define APPPATH_KEY      HKEY_LOCAL_MACHINE
#define APPPATH_SUBKEY    "Software\\Microsoft\\Windows\\CurrentVersion\\App
Paths\\MYPROG.EXE"
#define APPPATH_CALL      _MYPATH\\MYPROG.EXE"
#define APPWORKDIR        _MYPATH"

```

```

HKEY hAppPath;
DWORD dwDisposition;

// Generate Quickstart key -----
if( RegCreateKeyEx( APPPATH_KEY,
                  APPPATH_SUBKEY,
                  0,
                  "",
                  REG_OPTION_NON_VOLATILE,
                  KEY_ALL_ACCESS,
                  NULL,
                  &hAppPath,
                  &dwDisposition ) == ERROR_SUCCESS )
{
    // Default value of key = path and name of EXE file —
    if( RegSetValueEx( hAppPath,
                      NULL,
                      0,
                      REG_SZ,
                      APPPATH_CALL,
                      lstrlen ( APPPATH_CALL) ) == ERROR_SUCCESS ) ;

    // Load PATH key value with start directory
    if( RegSetValueEx( hAppPath,
                      "Path",
                      0,
                      REG_SZ,
                      APPWORKDIR,
                      lstrlen( APPWORKDIR ) ) == ERROR_SUCCESS ) ;
}

```

Deleting keys

Any key that has been created can also be deleted (except for predefined keys such as HKEY_CURRENT_USER, etc.). The function for this is RegDeleteKey().

```

LONG RegDeleteKey (
    HKEY    hKey, // Key handle of higher level key
    LPCSTR  lpSubKey
);

```

The starting point for the delete process is a previously opened key (key handle), given by the parameter hKey. In order to delete a key you need the key handle of the key above it. In this case hKey can also be a predefined key such as HKEY_CURRENT_USER etc., if the key to be deleted is directly below it.

The name of the key itself is given by the parameter lpSubKey, which points to a C-string containing the name. Remember, all keys below this key will automatically be deleted as well, so you can delete an entire tree of keys by calling RegDeleteKey() once for the highest-level key. The rest is taken care of by the Registry. If no problems arise the function returns a value of ERROR_SUCCESS.

The following section of code demonstrates a call to RegDeleteKey(). Here we are deleting the key used for calling MYPROG.EXE via Start/Run, under HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\App Paths.

```

#define APPPATH_KEY    HKEY_LOCAL_MACHINE

```

```
#define APPPATH_SUBKEY    "Software\\Microsoft\\Windows\\CurrentVersion\\App
Paths\\MYPROG.EXE"

if ( RegDeleteKey( APPATH_KEY, APPPATH_SUBKEY ) == ERROR_SUCCESS )
    printf( "_Key deleted\\n")
```

Deleting key values

Besides the keys themselves you can also delete specific key values. First, you need the handle of the key in question. You then pass this handle to `RegDeleteValue()`, the function responsible for deleting key values.

```
LONG RegDeleteValue (
    HKEY    hKey, // Handle of key whose key value is being deleted
    LPCSTR  lpValueName // Pointer to C-string with name of key value
);
```

The second parameter is a pointer to a C-string containing the name of the key value to be deleted. To delete an unnamed key value, enter `NULL` instead of an actual pointer.

If the desired key value was deleted, the function returns `ERROR_SUCCESS`.

Loading and saving complete subtrees

Besides functions for accessing individual keys and key values, the Registry API has four functions which let you save complete subtrees to external files and from there, reload them. You can save parts of a Registry or add large sections to it without having to create all the keys and key values manually. Create the subtree once, save it in a file and load the entire subtree at runtime (or installation) of an application. This saves much work.

Saving subtrees

The basis for all these functions is `RegSaveKey()`. It can write a complete subtree of keys to an external file. Only files that have been created with `RegSaveKey()` can later be reloaded into the Registry.

```
LONG RegSaveKey (
    HKEY    hKey, // Key handle of starting point
    LPCSTR  lpFile, // Pointer to C-string with filenames
    LPSECURITY_ATTRIBUTES lpSecurityAttributes // Win 95: NULL
);
```

The starting point required by `RegSaveKey()` is the key handle of the key at the beginning of the subtree to be saved. This can be one of the predefined keys or a key generated by `RegOpenKey()` or `RegCreateKey()`. The `lpFile` parameter points to a character string naming the file to which the key and all its subkeys will be saved. The string can include a complete path name and a device name. It cannot, however, include a filename extension. Otherwise, the `RegLoadKey()` function will fail when you try to reload the saved entries. Also, the filename given must not already exist or the function will return an error.

When the given subtree is successfully saved, the function returns `ERROR_SUCCESS`. A look inside the saved file reveals that the Registry saves its entries not in ASCII but in binary format. An attempt to read the entries directly from the file or change the contents of the file will fail. To avoid this failure specifically classify the format for stored Registry files.

Loading subtrees

While `RegSaveKey()` allows you to save subtrees at any level, this is not the case with `RegLoadKey()`, which loads the subtrees. The keys in the file must be placed either directly below `HKEY_LOCAL_MACHINE` or `HKEY_USERS`. You can also place it exactly one level lower. Make certain it's one level lower and not two, three or four levels lower.

```
LONG RegLoadKey (
```

```

HKEY    hKey, // HKEY_USER or HKEY_LOCAL_MACHINE
LPCSTR  lpSubKey, // Pointer to C-string with name of subkey
LPCSTR  lpFile // Pointer to C-string with filenames
);

```

The key handle given by the first parameter must be one of the constants HKEY_LOCAL_MACHINE or HKEY_USERS. To insert another level between this one and the key being loaded, lpSubKey should point to a string giving the name of the intermediate level desired instead of an empty string. If this key doesn't yet exist, the function generates it. Finally, lpFile contains the address of a C-string with the name of the file to be loaded. If the keys from this file are successfully added to the Registry, the function returns a result of ERROR_SUCCESS.

Replacing subtrees

The function RegReplaceKey() is a combination of RegSaveKey() and RegLoadKey(). This function not only loads a subtree previously saved by RegSaveKey() but at the same time writes the overwritten keys and values to a kind of backup file. The old contents are thereby saved and can later be reloaded using ReadLoadKey() or a second call to RegReplaceKey().

```

LONG RegReplaceKey (
    HKEY    hKey, // HKEY_USERS or HKEY_LOCAL_MACHINE
    LPCSTR  lpSubKey, // Pointer to C-string with name of subkey
    LPCSTR  lpNewFile, // Pointer to C-string with filename being loaded
    LPCSTR  lpOldFile // Pointer to C-string with filename being saved
);

```

On the other hand, the starting point for the subtree to be replaced (hKey parameter) can only be HKEY_USERS or HKEY_LOCAL_MACHINE. Use lpSubKey to skip an extra level past these. Then enter a string with the name of the key in between the levels. The file used to load the entries is given by lpNewFile, a pointer to a string containing the filename. As before, this string can include a drive and path specification but no filename extension. The same also applies to lpOldFile, which gives the name of the backup file. If the entire operation runs successfully, the function returns ERROR_SUCCESS.

Unloading subtrees

Once you have loaded a subtree through RegLoadKey() or RegReplaceKey(), you can also unload it using RegUnloadKey().

```

LONG RegUnLoadKey (
    HKEY    hKey, // HKEY_LOCAL_MACHINE or HKEY_USERS
    LPCSTR  lpSubKey // Pointer to C-string with name of subkey
);

```

Again, hKey gives the origin of the subtree within the Registry. It must equal one of the two constants HKEY_LOCAL_MACHINE or HKEY_USERS. If the subtree to be unloaded doesn't begin directly below one of these two keys, specify its starting point through lpSubKey. It's a pointer to the name of the correct key, which itself must lie directly below HKEY_LOCAL_MACHINE or HKEY_USERS.

Sample programs

Two programs called PERSIST and REGSTAT will demonstrate the Registry and Registry API functions in more detail. PERSIST shows various ways you can use the Registry regarding application programs. The Registry in this program does the following:

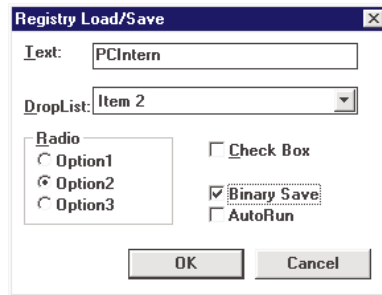
- Make user settings persistent
- Set up the program for autostart
- Enable uninstallation of the program through the Control Panel
- Allow the user to call the program using Start/Run

REGSTAT proceeds differently. This program goes through the entire Registry key by key and collects statistical information such as the number of keys, maximum level depth, longest key name, etc. Of course, a normal application program wouldn't have much use for this. However, it's a good example of a system utility program.

PERSIST

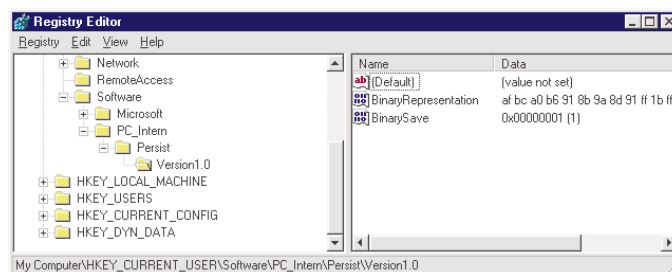
PERSIST shows how application programs can use the Registry. To the caller, the program presents itself as a dialog box with various input fields. The contents of these fields are saved permanently through the Registry. The next time the program is called, the fields display the most recent values entered.

PERSIST remembers the user's entries until the program is called again

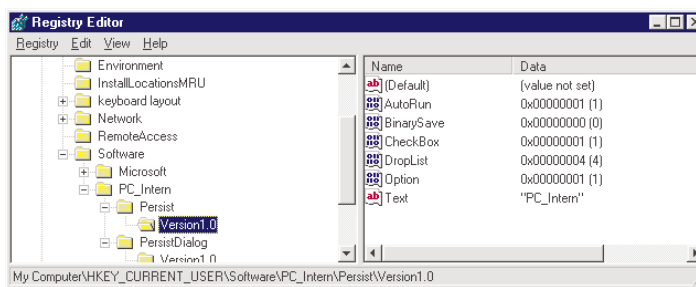


This program even remembers entries from different users logged onto the same machine. It does this by saving their respective Registry entries under HKEY_CURRENT_USER. As the following illustration shows, the program follows the default path and places its keys under HKEY_CURRENT_USER\Software\PC_Intern\PersistDialog\Version1.0.

Storing persistent settings as a binary structure



Storing persistent settings as individual key values

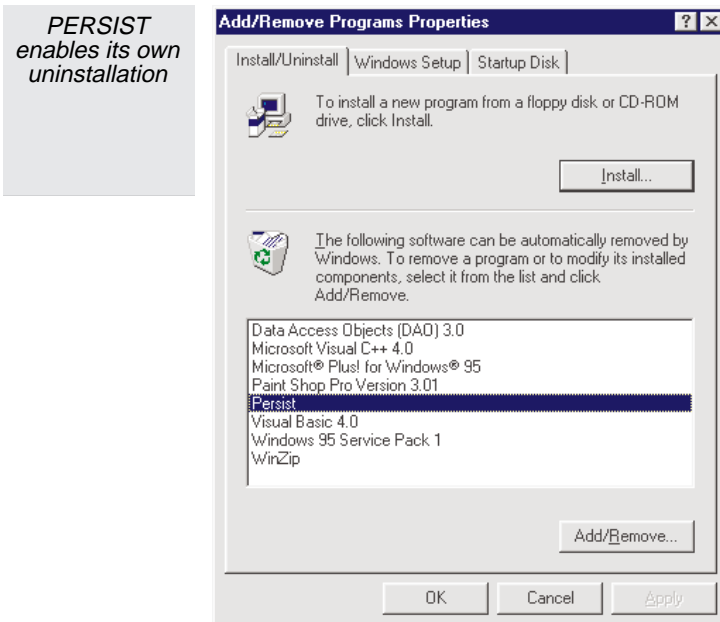


How the information is stored is decided within the dialog box under *_Save as Binary*". If the user activates this field, the individual data items are combined into a structure, encrypted and saved in the key value named BinaryRepresentation. At the same time the program sets the BinarySave key value to 1. This indicates the data has been stored as a structure.

On the other hand, the program works very differently if the user deactivates the *_Save as Binary*" checkbox. BinarySave is set to 0. A series of individual key values is saved instead of BinaryRepresentation. These represent the contents of the various fields in the dialog box.

To read the correct key values when the program starts, the program first inspects the contents of BinarySave. The program loads either the BinaryRepresentation key value or the individual key values for the dialog box fields. What the program loads depends on what it finds in the BinarySave contents.

PERSIST enables its own uninstallation by creating a key (along with the appropriate key values) under HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall. As the following illustration shows, PERSIST then appears in the Control Panel under _Add/Remove Programs_. If the user selects this program for uninstallation, it is called with the command line switch /u. This doesn't uninstall the program but sends a message to the user on the screen.



If the user enables the AutoRun checkbox, PERSIST adds a key value under HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run containing the program's path. The next time Windows 95 starts, the program starts automatically as well. The following listing of PERSIST.C shows how this works in practice. It contains several routines that you can use in your programs to store persistent data with very little overhead. This applies especially to the functions SetAppPath(), SetAutoRun() and SetUninstall().

The encryption of binary data, by the way, occurs through a very simple mechanism in the XORData() function. The individual bytes are simply negated, i.e., all bits are reversed through an XOR with 0xFF. Although this method won't stop an experienced hacker, the information won't be available to an average user. Also, it makes the decoding process very easy for the software because the negation is reversible. All it does is call the XORData() function a second time for the same data.

RegStat

REGSTAT demonstrates a recursive search through the Registry, as well as the use of RegQueryInfoKey(). When called with no command line arguments, the program returns the following information:

```
c:>REGSTAT
```

```
REGSTAT - (c) 1995 by Michael Tischer & Bruno Jennrich
```

```
Maximum level depth      : 10
Total number of keys     : 13911
Total number of key values : 18857
```

```
Maximum length of key name      : 59
Maximum length of value name   : 68
Maximum key value               : 2093
```

Total number of key value types

```
Binary data                    : 2456
DWORDs                        : 896
Big Endian DWORDs             : 0
Strings                       : 15478
String arrays                 : 15
Strings with environment variables: 12
```

You'll find the following program(s) on the companion CD-ROM



PERSIST.C (C listing)

Although REGSTAT will return a different number of keys on your machine, their magnitudes may be identical. Almost twenty thousand key values - no one can say Windows 95 can't keep track of settings. If you add another argument on the command line, REGSTAT interprets this argument as the name of a key for which it should look. It still searches the entire Registry, but instead of the statistical information it lists all occurrences of the given key. The following illustration shows a search for the key `_fonts` and how many times it was found.

```
C:>REGSTAT fonts
```

```
REGSTAT - (c) 1995 by Michael Tischer & Bruno Jennrich
```

```
Keys: HKEY_CURRENT_USER\Software\Microsoft\Visual C++ 2.0\Fonts
```

```
Keys: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Fonts
```



```
Keys: HKEY_LOCAL_MACHINE\Config\0001\Display\Fonts
```

```
Keys: HKEY_USERS\Default\Software\Microsoft\Visual C++ 2.0\Fonts
```

```
Keys: HKEY_CURRENT_CONFIG\Display\Fonts
```


Index

Symbols

	437–438
 key status	263–264
-ACKnowledge line	255
-STROBE line	255
80286 DOS extensions	543–544
80286 Protected Mode	514–527
See also Protected mode	
80286 registers	515
80386	
DOS extensions	544–546
Protected mode	527–536
80386 DOS extensions	
Flat model	545
8253 timer	324–325
8259A Programmable Interrupt Controller	
See also PIC	207
8514/A	80

A

Access times	
Hard drives	354–355
Address register	15
Addressing modes	534
ADSR parameters	605–606
Aliasing	523–524
ANSI.SYS	440
ASCII character set	83
ASCII codes	270
Assembly language	
COM programs	393–396
ASSIGN	440
Asynchronous communication	232–234
AT (Advanced Technology)	10
AT bus	4–5
AT hardware interrupts	23
AT Realtime clock	
Accessing registers	362–364
Accessing with functions	360–361
Battery operated RAM	365–366
Clock registers	361–364
Configuration data	365–366
Functions	360–361
Interrupt 08H	359–360
Reading	361–364
Setting	361–364
AT realtime clock	417–418
AT ROM-BIOS	359–361
Attribute controller	
EGA/VGA controllers	187–190

B

Batch child	
Using EXEC function	434
Batch files	388–389
Battery operated RAM	
AT realtime clock	365–366
BIOS	63–75
Accessing disk drives	332–339
Accessing extended memory ..	312–314
Accessing hard drives	339–342
Accessing serial ports	246–248
Accessing the keyboard	268–282
BIOS architecture	63–64
BIOS hard drive interrupt	339–340
BIOS Standard	63–64
Color selection for EGA/VGA cards ...	
.....	157–158
Determining versions	66–67
ESDI controllers	344–345
Functions	251–252, 312–314
Hard drive parameters	64
Importance of BIOS	63
Joysticks (reading)	289–290
Parallel port (accessing)	249–252
POST	64–66
Printer interrupt	249–252
RAM configuration (determining) .	369
Reading date and time	359–361
SETUP program	64
Shadow RAM	64
System configuration (determining)	
.....	367–369
Time out error	249–250
Variables	67–75
Versions	66–67
XT BIOS variables	73–75
BIOS interrupts	
AT realtime clock	359–361
RAM configuration (determining) .	369
System configuration	367–368
BIOS keyboard handler	26
BIOS keyboard interrupt	26
Interrupt 16H	282–283
Interrupt variables	278–280
Keyboard buffer	279–280
Keyboard buffer location	280
BIOS printer interrupt	251–252
BIOS Standard	63–64
BIOS variables	67–75
BIOS-proof keys	273–274
Bitplanes	136–145
Dividing video RAM	136
Graphics controller	137–138
Latch registers	137

Read mode 0	138–139
Read mode 1	139–141
Write mode 0	141–143
Write mode 1	143
Write mode 2	144
Write mode 3	144–145
Blaster environment variable	601
Block device drivers	441
Block level protocol	
Parallel port	263–264
Boot files	389
Boot partition	357
Bootstrap process	66
Bootstrap routine	379
Branch Target Buffer	30–31
Break flag	437
Buffer addresses	
Transferring	540–542
Buffers	
C language	59–61
Turbo Pascal	51–53
Bus	
See specific type	
BUSY line	255

C

C language	54–62
Buffers	59–61
Data types	54–57
Interrupts, calling	57–59
Port accesses	62
Cache controllers	355
Cache programs	355
CAV (Constant Angular Velocity) 561–562	
CD-DA format (CD Digital Audio)	563
CD-ROM	559–600
Accessing with MCXDEX.API	
.....	573–579
CAV (Constant Angular Velocity)	
561–562	
CD-ROM driver	579–598
CD-ROM formats	564–565
CLV (Constant Linear Velocity)	
.....	561–562
Data organization	561–567
Dimensions	560
Error correction	564
Formats	559–571
Frames	562–563
High Sierra format	567–571
Integrating into DOS/Windows	
.....	571–573
Lands	560
MCXDEX.API functions	574–579
Mode 1 format	564–565
Mode 2 format	564–565
Photo CD	567–568
Physical format	559–567
Pits	560
Sectors	563
Storage capacity	564

- Storing bits/bytes 562
- Storing data 561–562
- Structure 560
- Subchannels 563–564
- Subcodes 563–564
- Technology 559–600
- XA format 565–566
- Yellow Book 564
- CD-ROM driver 579–598
 - Device drivers 581
 - Driver functions 581–582
 - Driver functions listed 583–598
 - Status word 582–583
- CD-ROM formats 559–571
 - Data organization 561–567
 - Directories/subdirectories 568–569
 - High Sierra format 567–571
 - High Sierra format files 568
 - High Sierra format logical blocks .. 568
 - High Sierra format sectors 568
 - High Sierra path table 569–570
 - High Sierra volumes 571
 - High Sierra XAR 570–571
 - Mode 1 format 564–565
 - Mode 2 format 565
 - Photo CD 567–568
 - Physical format 559–567
 - XA format 565–566
- CD-ROM, Storing data
 - Bits/bytes 562
 - CAV (Constant Angular Velocity) 561–562
 - CLV (Constant Linear Velocity) 561–562
 - Error correction 564
 - Frames 562–563
 - Sectors 563
 - Storage capacity 564
 - Subchannels 563–564
 - Subcodes 563–564
- CD-WO (CD-Write Once) 567–568
- Cell envelope 604–606
- Channels (Sound)
 - Percussion mode 612–613
 - Setting 610–613
 - Setting channel parameters 611
 - Setting parameters 611–612
- Character device drivers 440
- Character input
 - DOS functions 401–402
 - Handle functions 399–401
 - keyboard 400
 - Screen 400
 - Serial interface 401
- Character output
 - DOS functions 401–402
 - Handle functions 399–401
 - Keyboard 400
 - Screen 400
 - Serial interface 401
- Character table structure
 - EGA/VGA cards 123–125
- Child program 433
 - Using EXEC function 433
- Clock drivers 454
- Clock registers
 - AT realtime clock 361–364
- CLV (Constant Linear Velocity) 561–562
- Code accesses 520–521
- Code mutexes 689–691
- Color monitors
 - And video cards 91–92
- Color/Graphics Adapter 108–114
 - Attribute bytes 108
 - Bit layout 111–112
 - CRT controller 113–114
 - Flickering 112
 - Graphics modes 108
 - Registers 111
 - Resolutions 109–111
 - Text modes 108, 112
- Color/Graphics Adapter (CGA) 77–78
- COM files 391–398
 - Assembly language 393–396
 - Differences from EXE files 391–392
 - General information 392–395
 - Memory limits 393
 - Microsoft support 392
- Command processor 387–388
- Common Controls 777–866
 - Image List 778–791
 - Image List, Adding images 782–783
 - Image List, Characteristics 778–780
 - Image Lists, Creating and destroying .. 780–781
 - Image Lists, Drag and Drop ... 787–789
 - Image Lists, Drawing images . 783–784
 - Image Lists, Information on images 786–787
 - Image Lists, Merging images 785
 - Image Lists, Overlays 786
 - Image Lists, Querying and setting items 802–803
 - Image Lists, Reading and writing 789–790
 - Image Lists, Removing and replacing . 784–785
 - Image Lists, System Image List 790
 - Image Lists, Using 791
 - List Views 826–866
 - List Views, Adding and deleting items 833–837
 - List Views, Browsing and searching ... 843–846
 - List Views, Colors and fonts .. 842–843
 - List Views, Communication .. 826–829
 - List Views, Drag and Drop 859–866
 - List Views, Editing text 852–853
 - List Views, Generating 829–833
 - List Views, Images 837–839
 - List Views, Mouse/keyboard events ... 856–859
 - List Views, Positioning items 850–851
 - List Views, Querying 839–842
 - List Views, Report view 853–856
- List Views, Setting 839–842
- List Views, Setting texts and images ... 851–852
- List Views, Setting visible area 846–850
- Tree Views 791–826
- Tree Views, Browsing 806–809
- Tree Views, Drag and Drop ... 819–825
- Tree Views, Generating during runtime 795–802
- Tree Views, Mouse/keyboard events ... 817–819
- Tree Views, Opening and closing 805–806
- Tree Views, Owner-draw 816
- Tree Views, Processing a label 811–814
- Tree Views, Selecting the current item 809–811
- Tree Views, Setting graphics . 803–805
- Tree Views, Sorting 814–816
- TreeViews, Communication .. 792–795
- Common registers 13–14
- Compressed Volume File (CVF) 483–484
- Continuous envelope 606
- Control codes (keyboard) 269
- Control Panel 932–933
- Controllers
 - Clock (8248) 7
 - CRT controller (6845) 7
 - Disk controller (765) 8
 - Disk drives 331
 - DMA controller (8237) 6
 - DMA controllers 217–229
 - Hardware interrupt 6–7
 - Interrupt controller (8259) 6
 - Math coprocessors 8–9
 - Programmable peripheral interface 7
- Cooked mode 441
 - Switching between raw mode 402–403
- Critical error handler 438
- Critical error interrupts 437–438
- Critical sections 688–691
- CRT controller 92–93
 - Accessing with MDA card 103–104
 - Color/Graphics Adapter (CGA) 113–114
 - EGA/VGA registers 175–184
 - Hercules Graphics Card 105–106
 - Registers 93–95
 - Scrolling with EGA/VGA cards 131–132
- CRT controller (6845) 7
- CRTC 92–93
- CT1335 mixer 639–640
- CT1345 mixer 640–644
- CVF (Compressed Volume File) 483–484
 - Clustering compressed files 485
 - Data structures 485–488
 - File structure 484–485
- Cylinder skewing 351–352

D

- DAC color table 156–157
- Data compression 481–494
 - Huffman coding 482–483
 - LZ process 483
 - Run Length Encoding 481–482
- Data lines register 254
- Data transfer at byte level 260–261
- Date
 - Reading from BIOS (AT realtime clock) 359–361
- Device attributes (Device drivers) 442–443
- Device drivers 439–462
 - Calls from DOS 454
 - CD-ROMs 460–462
 - Clock drivers 454
 - Developing 456–459
 - Device driver access 441
 - Examples of drivers 456–459
 - EXE programs 460
 - IOCTL 454–456
 - Structure 442–444
- Device drivers, DOS 439–462
 - See also DOS device drivers
- Device drivers, Functions 444–453
 - Function 00H: Driver initialization 445
 - Function 01H: Media check ... 446–454
 - Function 02H: Build BIOS Parameter Block (BPB) 447–454
 - Function 03H: I/O Control Read 448–454
 - Function 04H: Read 448–454
 - Function 05H: Non-destructive Read .. 449–454
 - Function 06H: Input Status 449–454
 - Function 07H: Flush Input Buffers 449–454
 - Function 08H: Write 450–454
 - Function 09H: Write with Verify 450–454
 - Function 0AH: Output Status . 451–454
 - Function 0BH: Flush Output Buffers ... 451–454
 - Function 0CH: I/O Control Write 451–454
 - Function 0DH: Open 452–454
 - Function 0EH: Device Close .. 452–454
 - Function 0FH: Removable Media 452–454
 - Function 10H: Output Until Busy 453–454
 - Function 17H: Get Logical Device 453–454
 - Function 18H: Set Logical Device . 453
- Device drivers, IOCTL 454–456
- Device drivers, Structure 442–444
 - Device attributes 442–443
 - Device header 442
 - Field structure 443
 - Interrupt routines 443
 - Strategy routines 443
- Device header 442
- Digital Sound Processor (DSP) ... 615–638
 - 8-bit mono, auto init DMA mode 627–628
 - 8-bit mono, high speed auto init DMA mode 628
 - 8-bit mono, high speed single cycle DMA mode 628
 - 8-bit mono, single cycle DMA mode ... 627
 - 8-bit stereo, high speed auto init DMA mode 629–637
 - 8-bit stereo, high speed single cycle DMA mode 628–629
 - Accessing 621–624
 - Command reference 629–638
 - Commands by function group 624–625
 - DMA controller 619
 - Double buffering 619–620
 - Handling interrupts 623–624
 - Reading data from DSP 621
 - Resetting DSP 622
 - Sampling frequency 625–626
 - Sampling rate 625–626
 - Sending data/commands to DSP 621
 - Speaker on/off 626
 - Transfer length 626
 - Transfer modes 617–618
 - Version (determining) 625
 - Version 4.xx commands 626–629
 - Versions of DSP 616–617
- Digital to Analog Converter (DAC) 194–197
- Directories
 - Accessing in DOS 411–416
- Directory tree 411
- Disk controller (765) 8
- Disk Drive Parameter Table 335–336
- Disk drives 329–332
 - 3.5-inch drives 330–331
 - 5.25-inch drives 329–330
 - Accessing with BIOS 332–339
 - Controllers 331
 - Disk Drive Parameter Table ... 335–336
 - Programming the controller ... 335–336
- Disk drives, BIOS functions
 - Disk Drive Parameter Table ... 335–336
 - Drive status 332, 334–335
 - Prompt for drive type 333
 - Reading diskette sectors 334
 - Resetting drive 333
 - Verifying diskette sectors 334
 - Writing drive sectors 334
- Diskettes
 - 3.5-inch diskettes 330–331
 - 5.25-inch diskettes 329–330
 - Characteristics 327–329
 - Formats 329–332
 - Structure/appearance 327–328
- DLL (Dynamic Link Libraries) ... 757–776
 - Address space of a process 769–770
 - API Functions 762–764
 - Basics of DLLs 757–770
 - Global variables 769–770
 - Initializing 760–762
 - Linking DLLs 767–768
 - Load time dynamic linking 758–760
 - Multithreaded DLLs 770–776
 - Producing with C modules 764–767
 - Resource memory 770
 - Run time dynamic linking 758–760
 - Terminating 760–762
 - Thread Local Storage (TLS) .. 771–776
- DMA controller 619
- DMA controller (8237) 6
- DMA controller (Direct Memory Access) 217–229
 - 16-bit transfers 223
 - Channels 218–220
 - Disk drives using DMA controller 217–218
 - DMA segment overflow 221–222
 - Modes of operation 223–224
 - Priorities 218
 - Programming the DMA controller 224–229
 - Registers 220–221
 - XT/AT architecture 222
- DOS
 - Batch files 388
 - Boot files (searching for) 389
 - Booting DOS 389
 - Command processor 387–388
 - Components 387–388
 - Directories 411–416
 - DOS filters 403–404
 - DOS kernal 387
 - DOS-BIOS 387
 - Executing programs in Windows 473–474
 - Network programming 468–472
 - RAM management 419–421
- DOS 6.x 481–494
 - Data compression 481–494
 - DoubleSpace 481–494
- DOS device drivers 439–441
 - ANSI.SYS 440
 - ASSIGN 440
 - Block device drivers 441
 - Character device drivers 440
 - Cooked mode 441
 - Custom drivers 439
 - Device driver access 441
 - Raw mode 441
- DOS directories
 - Accessing 411–416
 - Creating 411
 - Directory tree 411
 - FCB functions 413–414
 - File attribute 412–413
 - Functions used in directories .. 411–416
 - Handle functions 414
 - Search function 412
 - Specifying 412

- DOS Extensions 539–546
 - 80286 DOS extensions 543
 - 80386 544–546
 - Buffer addresses 540–542
 - Calling DOS functions 546
 - Flat model 545
 - Operation 539
 - Protected mode 539–543
 - DOS extensions
 - Memory management 542
 - DOS file management
 - FSB functions 407–410
 - Handle functions 405–407
 - DOS file system 377–385
 - Basic structure 377–378
 - Boot sector 378–379
 - File Allocation Table (FAT) .. 379–382
 - Root directory 382–384
 - Subdirectories 384–385
 - DOS filters 403–404
 - Combined filters 404
 - Pipes 404
 - SORT 403
 - DOS functions 401–402
 - Character output 402
 - Keyboard 401–402
 - Printer output 402
 - Sensing Windows 473
 - Serial interface 402
 - DOS Info Block (DIB) 477–480
 - DOS buffer 479
 - Drive Parameter Block (DPB) 478–479
 - Path table 479
 - Structure 478
 - DOS kernel 387
 - DOS Protected Mode Interface (DPMI)
 - See DPMI
 - DOS RAM Management
 - Changing size of memory blocks ... 421
 - Determining available RAM 420
 - Function 58H subfunctions 423
 - Functions 419–421
 - Memory allocation . 419–420, 424–425
 - Memory allocation models 423
 - Memory Control Block 425–432
 - Memory sources 421–423
 - Releasing memory 420–421
 - Routine memory management 422–423
 - Upper Memory Blocks (UMBs) 421–423
 - Viewing memory allocation ... 424–425
 - DOS structures
 - Undocumented DOS structures 477–480
 - DOS-BIOS 387
 - Double buffering 619–620
 - DoubleSpace 481–494
 - Compressed Volume File (CVF) 483–484
 - File storage 486
 - MRCF 492
 - MRCI 492
 - DPMI 552–558
 - Client initialization 555–556
 - Clients/hosts 553
 - Debug registers 558
 - Differences from VCPI 552–553
 - DOS memory allocation 557
 - DPMI interrupt handling 558
 - DPMI outputs/functions 554–555
 - Extended memory allocation .. 556–557
 - Interrupt handling 557–558
 - Local descriptor table 556
 - Real mode routines (calling) .. 557–558
 - Services 554–555
 - Switching into protected mode 555–556
 - Virtual machine 553–554
 - Virtual memory management 557
 - Drive Parameter Block (DPB) 478–479
 - DPB access 479
 - Structure 478
 - DSP transfer modes 617–618
 - ADPCM DMA mode 618
 - Auto initialize DMA mode 617
 - High speed DMA mode 618
 - Single cycle direct mode 617
 - DTA (Disk Transfer Area) 409
 - DTA structure 414–416
 - Dynamic Link Libraries
 - See DLL (Dynamic Link Libraries)
- ## E
- EGA cards 115–196
 - 16 color graphics mode 146–150
 - Bitplanes 136–145
 - Changing pixel display 120–121
 - Character map structure 125–127
 - Character structure 118–120
 - Character table structure 125–127
 - Character wrapping 132–133
 - Color selection 155–160
 - Disabling screen 135–136
 - Displaying 512 characters 125–127
 - Displaying special characters . 127–128
 - EGA BIOS 118–120, 124–125
 - EGA monitors 115
 - Font loading/defining 118–120
 - Font selection/programming .. 117–129
 - Graphics controller 137–138
 - Graphics mode (16 colors) 146–150
 - Graphics mode fonts 129
 - Graphics window 127–128
 - Horizontal pel panning register 130–131
 - Identifying 116–117
 - Monitor selection 115–117
 - Pel panning register 129–131
 - Scrolling (smoother) 129–135
 - Scrolling in text mode 134–135
 - Scrolling screen contents 131–132
 - Sprites 160–173
 - Vertical pel panning register 131
 - EGA cards, 16-color graphics mode 146–150
 - Accessing individual pixels 147–150
 - EGA cards, 256-color graphics mode
 - Addressing pixels 155–156
 - EGA cards, Bitplanes 136–145
 - Dividing video RAM 136
 - Graphics controller 137–138
 - Latch registers 137
 - Read mode 0 138–139
 - Read mode 1 139–141
 - Write mode 0 141–143
 - Write mode 1 143
 - Write mode 2 144
 - Write mode 3 144–145
 - EGA cards, Color selection 155–160
 - DAC color table 156–157
 - Palette registers 155–156
 - Using BIOS to set color 157–158
 - EGA cards, Registers 173–196
 - Attribute controller 187–190
 - CRT controller registers 175–184
 - Graphics controller 190–194
 - Sequencer controller 184–187
 - EGA cards, Scrolling 129–135
 - Character wrapping 132–133
 - Coordinating with CRT controller 133–134
 - Horizontal pel panning register 130–131, 133–134
 - Horizontal scrolling 130
 - Pel panning register 129–131
 - Screen contents 131–132
 - Text mode 134–135
 - Vertical pel panning register 131
 - Vertical scrolling 131
 - EGA cards, Sprites 160–173
 - EGA monitors 115
 - EMM (Expanded Memory Manager) . 307
 - EMS emulators 536–537
 - Enhanced Graphics Adapter (EGA) 78–79
 - Enhanced Small Devices Interface 344
 - Environment block
 - EXEC function 434–436
 - Error correction
 - CD-ROM 564
 - Hard drives 353
 - ESDI controller 344–345
 - Hardware configuration 344–345
 - Event objects 696–700
 - EXE files 391–398
 - Differences from COM files 391–392, 395–396
 - General information 395
 - Memory allocation 396–398
 - EXE programs
 - See also EXEC function
 - Using as device drivers 460
 - EXEC function
 - Batch child 434
 - Child program 433–434
 - Environment block 434–436
 - Memory problems 436

- Parameter blocks 434
 - Parent program 433
 - Relocation factor 436
 - Expanded BIOS Parameter Block (MDBPB) 487
 - Expanded memory 306–311
 - Accessing memory 309
 - Errors when accessing 314
 - Expanded Memory Manager (EMM) 307
 - Expanded Memory System (EMS) 307–311
 - Functions 308–311
 - Global Descriptor Table 312–314
 - History 308
 - History and versions 308
 - LIM standard 307–311
 - Limits to EMS allocation 309–311
 - Memory window 307
 - Page frame division 307
 - EMS Version 3.2 308
 - EMS Version 4.0 308
 - Extended Attribute Records (XAR) 570
 - Extended keyboard codes 270
 - Extended memory 311–322
 - Accessing 312–314
 - Conflicts 315–316
 - Extended memory blocks (EMB) 319–322
 - Functions 312–314
 - HMA access 317–319, 321
 - HMA access from real mode .. 316–319
 - Upper memory blocks (UMB) 319–322
 - XMS standard 319–322
 - Extended Memory Blocks (EMB) 319–322
 - Copying 321
 - Releasing 321
 - Extended Memory Specification (XMS) .. 319–322
- F**
- FAT (File Allocation Table) 353, 379–382
 - Checking for multiple copies 382
 - Fat layout 380–382
 - File clusters 381
 - File fragmentation 379–380
 - FCB functions 407–410
 - DTA 409
 - Extended FCB 408–409
 - File access 409
 - File searches 413–414
 - Open FCB 409
 - File fragmentation 379–380
 - Flickering (CGA cards) 112
 - Floating-point Unit
 - Pentium processor 37–38
 - FM method (Hard drive recording) 348
 - FM synthesis 602–614
 - Physical nature of sound 602–604
 - Fonts
 - EGA graphics mode 129
 - Selecting/programming cards . 117–129
 - VGA graphics mode 129
 - Foreign language keyboards 266
 - Frequency Modulation
 - FM programming 613–615
 - Principle of FM 603–604
 - See also FM synthesis
 - Functions
 - AT realtime clock 360–361
 - BIOS functions (printing) 251
 - Parallel port and BIOS 249–252
 - RAM management (DOS) 419
- G**
- Gates 520–521
 - Generator cells
 - Setting 608–613
 - Global Descriptor Table 312–314
 - Global descriptor tables 521–522
 - Graphics controller 137–138
 - EGA/VGA registers 190–194
 - GUID (Globally Unique Identifier) .. 873–874
 - Interface declaration 877–880
 - OLE functions and classes 880–888
 - Structure 874
- H**
- Handle functions 399–401, 405–407
 - Access length 406
 - Advantages over FCB functions 410
 - Device names 399–401
 - File searches 414–416
 - Keyboard 400
 - Screen 400
 - Serial interface 401
 - Hard drive controllers 342–347
 - Data transfer rates 342
 - ESDI controller 344–346
 - IDE controller 345–346
 - SCSI controllers 345
 - ST506 controller 343
 - Hard drive partitions 355–358
 - Boot partition 357
 - Formatting process 355
 - Partition sector 356
 - Partitioning 355–356
 - Hard drives
 - Access times 354–355
 - Accessing with BIOS 339–342
 - BIOS hard drive interrupt 339–340
 - Boot partition 357
 - Cache controllers 355
 - Cache programs 355
 - Characteristics 327–329
 - Controllers 342–347
 - Cylinder skewing 351–352
 - Data transfer rates 342
 - Determining hard drive parameters 341
 - Determining status 340
 - Error correction 353
 - ESDI controller 344–346
 - Extended sector read/write 342
 - Extra tracks 354
 - FM method of recording information 348
 - Formatting 355–358
 - Formatting hard drive cylinders 341
 - ID field 353
 - IDE controller 345–346
 - Increasing capacity 353
 - Initializing unknown hard drives ... 342
 - Interleave factor 350–351
 - Interleaving 350–351
 - MFM method of recording information 348–349
 - Multiple zone recording 353
 - Partition sector 356
 - Partitioning 355–356
 - Partitions 355–358
 - Reading hard drive sectors 341
 - Recalibrating 342
 - Recording information 347–350
 - Resetting hard drive controller 340
 - RLL method of recording information 349–350
 - SCSI controller 345
 - Setting interleave factor 351
 - ST506 controller 343
 - Status code 340
 - Structure/appearance 328–329
 - Testing controller 342
 - Tracks 351–352
 - Verifying hard drive sectors 341
 - Writing hard drive sectors 341
 - Hard drives, BIOS functions 339–342
 - BIOS hard drive interrupt 339–340
 - Determining hard drive parameters ... 341
 - Determining status 340
 - Extended sector read/write 342
 - Formatting hard drive cylinders 341
 - Initializing unknown hard drives ... 342
 - Reading hard drive sectors 341
 - Recalibrating 342
 - Resetting hard drive controller 340
 - Status code 340
 - Testing controller 342
 - Verifying/writing sectors 341
 - Hardware interrupts 6–7
 - AT hardware interrupts 24–25
 - PC hardware interrupts 23
 - PC/XT hardware interrupts 23
 - XT hardware interrupts 23
 - Hercules Graphics Card 78, 104–107
 - CRT controller 105–106
 - Graphics mode 106–107
 - Video RAM 104–105
 - Hertz 323
 - High Memory Area (HMA) 316–319
 - Direct access from real mode . 316–319
 - High mute 607

High Sierra CD-ROM format 567–571
 Directories/subdirectories 568–569
 Files/filenames 568
 Logical blocks/sectors 568
 Path table 569–570
 Volumes 571
 XAR 570–571
 HMA access 317
 Horizontal pel panning register 130
 Horizontal synchronization
 MDA card 103
 Huffman coding 482–483

I

I/O access interception 538
 I/O ports 253
 IDE controller 345–346
 Image List 778–791
 Adding images 782–783
 Characteristics 778–780
 Creating and destroying 780–781
 Drag and Drop 787–789
 Drawing images 783–784
 Image information 786–787
 Loading entire bitmaps 781–782
 Merging images 785
 Overlays 786
 Reading and writing in streams
 789–790
 Removing and replacing images
 784–785
 System Image List 790
 Using Image Lists 791
 In Service Register (ISR) 209
 Intel 80xx series 11–13
 Intelligent Drive Electronics 345
 Interleave factor 350–351
 Interlocked variables 700–701
 Interrupt 16H functions 268–282
 Interrupt 27H 463–466
 See also Multiplexer
 Interrupt controller
 See also PIC
 Interrupt controller (8259) 6
 Interrupt Descriptor Table (IDT) 525–527
 Interrupt Mask Register (IMR) 209
 Interrupt mode 238
 Interrupt Request Register (IRR) 209
 Interrupt vector table 20–23
 Interrupts 20–25
 C language 57–59
 Data and time 359–361
 Disabling hardware interrupts 23–24
 Hardware interrupts 23–28
 Interrupt vector table 20–23
 Protected mode 525–527
 Software interrupts 20
 Turbo Pascal, calling in 50–51
 IOCTL function 454–456
 ISA bus 5

J

Joysticks 289–291
 Connecting 289
 Determining position 289–290
 Reading buttons 290–291
 Reading from BIOS 289–290

K

Keyboard 265–287
 Accessing from BIOS 268–282
 ASCII codes 270
 BIOS-proof keys 273–274
 Control codes 269
 DOS functions 401–402
 Event handles 400
 Extended keyboard codes 270–273
 Foreign language keyboard 266–287
 Hardware components 25
 Interrupt 16H functions 268–282
 Keyboard handler 266–267
 MF-II keyboards 275–282
 Operation 265–267
 Programming basics 265–268
 Scan code 265–267
 Types of keyboards 267–268
 Keyboard buffer 26
 Keyboard buffer location 280
 Keyboard controller 284–287
 Keyboard handler 266–267
 Keyboard hardware interrupts 283
 Keyboard interrupt handler 282–283
 BIOS keyboard interrupt 16H 282–283
 Keyboards
 BIOS keyboard interrupt variables
 278–280
 Keyboard controller 284–287
 Keyboard interrupt handler 282–283
 LEDs 287
 Redirecting keyboard hardware
 interrupts 283
 Scan codes 280–282
 Switching on/off LEDs 287
 Typematic rate 285–287

L

Latch registers 137
 Lempel-Ziv process 483
 LIM standard 307–311
 List Views 834–866
 Adding and deleting items 833–837
 Browsing and searching 843–846
 Communication 826–829
 Drag and Drop 859–866
 Editing text 852–853
 Generating a ListView during runtime
 829–833
 Images 837–839
 Mouse/keyboard events 856–859
 Positioning and aligning items 850–851

Querying 839–842
 Setting colors and fonts 842–843
 Setting texts and images 851–852
 Setting the Image List set 839–842
 Setting visible area 848–850
 Working in report view 853–856

Local descriptor table

DPMI 556
 Local descriptor tables 521–522
 LZ process 483

M

Math coprocessors 8
 MCB chain 426–432
 MDA
 Control register 102–103
 CRT controller access 103–104
 Horizontal synchronization 103
 Status register 103
 Video RAM 102
 Memory
 EXEC function 436
 Expanded memory 306–311
 Extended memory 311–322
 Memory addresses 14–20
 Address register 15
 Offset addresses 15–16
 Segment addresses 15–16
 Segment register 15
 Memory allocation models 423
 Memory Control Block (MCB) 425–432
 Accessing MCB chain 426
 Managing allocated memory 425
 Structure 426
 Memory Controller Gate Array (MCGA)
 79–80
 Memory expansion
 Extended memory 311–322
 Extended memory conflicts 315–316
 XMS standard 319–322
 Memory management
 DOS extensions 542
 Protected mode 516–517
 Utilities 536–537
 Memory mapped files (MMF)
 See Virtual Memory Management
 MESI protocol 36
 MF-II keyboards 275–282
 BIOS keyboard interrupt variables
 278–280
 Extended scan codes 281
 Keyboard buffer 279–280
 New keyboard codes 275–277
 Status functions 277–278
 MFM method (Hard drive recording)
 348–349
 Microsoft DoubleSpace BIOS Parameter
 Block 487
 Microsoft Realtime Compression Format
 (MRCF) 492

- Mixer 638–645
 - Controlling 638–639
 - CT1335 mixer 639–640
 - CT1345 mixer 640–641
 - CT1745 mixer 641–644
 - Mixer registers 638–639
 - Programming 644–645
 - Modem communication protocol 236–237
 - Modem registers 243–244
 - Monitors
 - And video cards 91
 - Monochrome Display Adapter (MDA)
 - See also MDA 77, 101–104
 - Mouse 293–304
 - Access functions 301–303
 - Buttons 295
 - Changing cursor 303
 - Communication 304
 - Cursor 295–304
 - Cursor appearance 295
 - Functions 293–301
 - Mickey measurement 296
 - Mouse cursor 295–304
 - Appearance 295
 - Changing 303
 - Hardware specific 298
 - Size 298
 - Software specific 298
 - Mouse functions 293–301
 - Access functions 301–303
 - AssmHand handler 302–303
 - Interrupt method 299–301
 - Listed 296–301
 - Polling method 299
 - Reading external devices 299–301
 - MRCI (Microsoft Realtime Compression Interface) 492
 - MS-DOS 6
 - See also DOS 6.x 481–494
 - MSCDEX.API
 - Accessing a CD-ROM drive .. 573–579
 - Calling a function 573–574
 - CD-ROM driver 579–598
 - History 572–573
 - MSCDEX.API functions
 - Accessing a CD-ROM drive .. 574–579
 - CD-ROM specific information 577–578
 - Checking for a CD-ROM drive 574
 - Determining drive letters 575–579
 - Reading/writing 576–577
 - Multiple zone recording 353
 - Multiplexer 463–466
 - Operation 463–464
 - Using DOS commands 464–466
 - Using the multiplexer 464–466
 - Multiplication factor 607
 - Multitasking 514, 649–655
 - Cooperative multitasking 651
 - Multiple threads 652–653
 - Preemptive multitasking 651
 - Priorities and scheduling 653–655
 - Priority classes 653–654
 - Sample programs 676–679
 - Thread priorities 654
 - Threads 650–653
 - Timeslice processing 650–653
 - Utilities 537–538
 - Multithreaded DLLs 770–776
 - See also DLL (Dynamic Link Libraries)
 - Mute factor 606
 - Mutexes 683–688
 - Functions 687–688
- ## N
- Network programming 467–472
 - Basic information 467–468
 - Function interfaces 468
 - IPX/SPX interface 468
 - Network operating system 467
 - Record locking 469
 - SQL servers 467
 - Under DOS 468–472
 - Novell NetWare 467
- ## O
- OCW 213–214
 - OLE (Object Linking and Embedding)
 - Development 867–888
 - DLL successor 868–880
 - Dynamic binding 870–871
 - GUIDs 873–874
 - Interface functions 871–872
 - Interface identification 873–874
 - Interfaces / implementation 869–870
 - OLE and C++ 872–873
 - UNICODE conversion 882–886
 - Operating system privileges 514
 - Operator cell (Sound generation) 604
 - Operator cells
 - Setting 609–613
 - OPL chip 608
 - Accessing internal registers 608–609
 - Orange Book 567
 - Overscan 93
- ## P
- Page frame division 307
 - Page tables 531
 - Dividing 532
 - VCPI 548–549
 - Page unloading strategies 533
 - Paging 529–531
 - Palette registers 155–156
 - Parallel port 249–264
 - ACKnowledge line 255
 - STROBE line 255
 - [Esc] key status 263–264
 - Accessing thru BIOS 249–252
 - Assigning names 253
 - Block level protocol 263
 - BUSY line 255
 - Cable connections 257
 - Cables 256–264
 - Communicating 255–256
 - Data lines register 254
 - Direct programming 252–264
 - Host control signals 256
 - I/O ports 253
 - Pins for parallel transfer cable 258
 - Port registers 253–255
 - Printer control register 254
 - Printer status register 254
 - Synchronization 262
 - Time out error 249–250
 - Time out problems 261–262
 - Transferring data 260–264
 - Parallel printer cables 256–264
 - Connection from port to printer 257
 - Pins for parallel transfer cable 258–264
 - Transferring data 260–264
 - Parallel transfer cable 258
 - [Esc] key status 263–264
 - Block level protocol 263
 - Byte level data transfer 260–261
 - Synchronization 262
 - Time out problems 261–262
 - Transferring data 260–264
 - Parameter blocks
 - EXEC function 434
 - Parent program
 - Using EXEC function 433
 - Partition sector 356
 - Partition table 356–357
 - PC development 3–10
 - AT 10
 - AT bus 4
 - Bus 4–6
 - Controllers 6–8
 - Memory layout 8–9
 - Processors 3
 - PS/2 10–11
 - Support chips 6–8
 - PC hardware
 - Basics 3–10
 - Bus development 4–6
 - Controllers 6–8
 - Memory layout 8–9
 - PC development 3–10
 - Processor development 3
 - Processors 11–18
 - Support chips 6–8
 - PC/XT hardware interrupts 23
 - PE (Portable Executable) format . 742–755
 - DOS header 744–745
 - Exports 753–754
 - File structure 742–744
 - Imports 751–753
 - Mapping to memory 743–744
 - Multiple views 748–751
 - PE header 745–747
 - Segment table 747–751
 - Peer to peer networking 468–469

- Pel panning register
 - Smoother scrolling EGA/VGA cards 129–131
 - Pentium processor 27–38
 - Associative caches 35
 - Block diagram 28
 - Branch Target Buffer 30–31
 - Cache 31–34
 - Features 28–29
 - First level cache 37
 - Floating-point unit 37–38
 - Levels of cache 31–32
 - MESI protocol 36
 - Pipeline execution 29
 - Program execution 29–31
 - Superscalar pipelines 30
 - Photo CD 567–568
 - PIC 207–216
 - Cascading 210–211
 - Commands 211–213
 - Connections 208–210
 - Initializing 211–213
 - Internal registers 209
 - Lines and wires 209
 - OCWs 213–214
 - Priority rotation 215
 - Pipeline procedure 29
 - Pipes 404
 - Polling 6
 - Polling mode 238
 - Ports 19
 - Addresses 19
 - Registers 253–255
 - POST (Power On Self Test) 64–66
 - Bootstrap process 66
 - Peripheral testing 65
 - ROM extensions 65
 - ROM modules 65–66
 - Tests described 64
 - Preemptive multitasking 514, 651–655
 - Printer
 - DOS functions for output 402
 - Printer status flags 250–251
 - Printer cables
 - Parallel cables 256–264
 - Printer control register 254
 - Printer interrupts
 - BIOS and parallel ports 249–252
 - Printer output
 - DOS functions 402
 - Printer status register 254
 - Printing
 - BIOS printer interrupt 251–252
 - Communication from computer to host 255–256
 - Parallel printer cables 256–264
 - Printer status 250
 - See also Parallel port
 - Setting printer in the Shell 929–932
 - Time out error 249–250
 - Priority rotation 215
 - Privileges 514
 - Privilege levels 518–520
 - Processes 655–670
 - See also *Synchronization*
 - As synchronization objects 701
 - Command line (accessing) 666
 - Ending a process 664–665
 - Environment variables 657–664
 - Exchanging data 668–670
 - Foreign processes 666–667
 - Inter-process communication 667–670
 - Predefined variables 662
 - Priorities (setting and reading) 665–666
 - Start function 661–662
 - Starting a process 656–662
 - Status information 666
 - Synchronization between processes 705–707
 - Synchronizing with threads 679–707
 - WIN32 functions controlling processes 655
 - Processor registers 12–14
 - See also *Registers*
 - Processors
 - See also *Pentium processor*
 - Determining processor type ... 370–375
 - Development 3
 - History/development 11–18
 - Intel 80xx series 11–13
 - Math coprocessors 8
 - Memory addresses 14–20
 - Pentium processor 27–38
 - Registers 12–14
 - Program Segment Prefix 391, 397
 - Memory locations 398
 - Structure and fields 397
 - Programmable peripheral interface (8255) 7
 - Protected mode 513–536
 - 80286 protected mode 514–527
 - 80286 registers 515
 - Addressing modes 534
 - Aliasing 523–524
 - Buffer addresses (transferring) 540–542
 - Characteristics 513–514
 - Code accesses 520–521
 - Descriptors (accessing) 522–523
 - DOS extensions 539–543
 - Gates 520–521
 - Global descriptor tables 521–522
 - Hardware access (restricting) 524
 - Interrupt Descriptor Table 525–527
 - Interrupts 525–527
 - Local descriptor tables 521–522
 - Memory management 516–517
 - Multitasking 514
 - Page table (dividing) 532
 - Page tables 531
 - Page unloading 533
 - Paging 529–531
 - Privilege levels 518–520
 - Privileges 514
 - Protective mechanisms 533
 - Requirements 513
 - Resource conflicts 546–558
 - Segment types 517–518
 - Selective I/O port blocking 533–534
 - Shadow registers 523
 - Switching to protected mode 516
 - Task switching 524–525
 - Task switching support 514
 - Utilities 536–538, 546–558
 - Virtual address size 522
 - Virtual memory 514–515
 - Protected mode (80386)
 - 80386 527–536
 - Addressing modes 534
 - General registers 529
 - New instructions 534
 - Registers 528–529
 - Segment descriptors 529
 - Selective I/O port blocking 533–534
 - Protected mode (80486)
 - 80486 527–536
 - Addressing modes 534, 536–537
 - General registers 529
 - Registers 528–529
 - See also Protected mode
 - Segment descriptors 529
 - Selective I/O port blocking 533–534
 - Virtual memory management . 529–531
 - Virtual-86 mode 535–536
 - Protected mode utilities
 - DPMI 546–558, 552–558
 - VCPI 546–558, 547–552
 - Protected mode utilities 536–538
 - Multitaskers 537–538
 - PS/2 system 10
- ## Q
- QuickBASIC
 - Accessing bit fields 41–43
 - Calling interrupts 43–45
 - Data types 39–43
 - Strings for DOS/BIOS functions 40–41
 - Structures and arrays 41
- ## R
- Race condition 680–682
 - RAM Management
 - Changing size of memory blocks ... 421
 - Determining available RAM 420
 - DOS RAM management 419–421
 - Memory allocation . 419–420, 424–425
 - Memory allocation models 423
 - Memory Control Block 425–432
 - Memory sources 421–423
 - Releasing memory 420–421
 - Routine memory management 422–423
 - Upper Memory Blocks 421–423
 - Viewing memory allocation ... 424–425
 - Raw mode 441
 - Switching between cooked mode 402–403

- Read mode 0 138–139
 - Read mode 1 139–141
 - Reed-Solomon-Code 564
 - Refresh rate 92
 - Registers
 - Address registers 15
 - Common registers 13–14
 - Development 12–14
 - DMA controller 220–221
 - Flag registers 14
 - Groupings 13
 - Offset addresses 15–16
 - Segment addresses 15–16
 - Segment register 15
 - Registry 951–975
 - Accessing keys and their values 958–970
 - Accessing through Win32 API 957–971
 - Loading and saving complete subtrees 970–971
 - Organization and structure 951–957
 - Saving application data 953–957
 - Relocation factor
 - EXEC function 436
 - RLL method 349–350
 - Root directory 382–384
 - RS-232 Standard 234–237
 - Run Length Encoding 481–482
- S**
- Sampling 614–638
 - Scan codes 280–282
 - MF-II extended scan codes 281
 - Screen
 - Event handles 400
 - Screen display
 - Disabling 135–136
 - Screen output
 - DOS functions 402
 - Screen pages
 - Multiple screen pages 99
 - Scrolling (screen)
 - EGA/VGA cards 129–135
 - SCSI controller 345
 - Sectors 377
 - CD-ROM 563
 - Segment register 15
 - Selective I/O port blocking 533–534
 - Semaphores 691–696
 - Controlling overflow 693–696
 - Sequencer controller
 - EGA/VGA registers 184–187
 - Serial interface
 - DOS functions 402
 - Event handles 401
 - Serial ports 231–248
 - Accessing in BIOS 246–248
 - Accessing UART registers 239
 - Asynchronous communication 232–234
 - Current UART register status 241–243
 - General information 231–232
 - Initializing UART registers 239
 - Interrupt mode 238
 - Modem communication protocol 236–237
 - Modem registers 243–244
 - Polling mode 238
 - RS-232 standard 234–237
 - Structure/appearance 237–246
 - Synchronous communication 232–234
 - UART registers 237–238
 - Serial transmission
 - RS-232 standard 235
 - SETUP
 - BIOS configuration 64
 - Shadow RAM in BIOS 64
 - Shadow registers 523
 - Shell
 - Control Panel 932–933
 - Copying and other file operations 900–902
 - Desktop items 893–897
 - Dialog boxes 933–941
 - Drag and Drop from the desktop 941–943
 - File operations 900–905
 - Finding the executable file for a document 906
 - Internet shortcuts 945–948
 - IShellFolder interface 908–918, 942–943
 - Namespace 890–892
 - Namespace structure 890–891
 - Naming ShellFolder Objects 891–892
 - Printers 929–932
 - Reading an icon from a file 905
 - Selecting folders 897–900
 - Shell API 892–908
 - Shell objects 902–905
 - ShellFolder objects 908–918, 926–929
 - Shortcuts 945–948
 - Taskbar Notification Area (TNA) 943–945
 - Updating the Namespace 907–908
 - Small Computer System Interface 345
 - Software interrupts 20
 - Sound 323–326
 - 8253 timer 324–325
 - Cycles 323
 - Demonstration programs 325
 - Generating on your PC 323–324
 - Ports 325
 - Timer frequencies 325
 - Sound Blaster cards 601–648
 - See also Sound generation
 - Blaster environment variable 601–602
 - Digital Sound Processor 615–638
 - History/development 601–602
 - Sound generation
 - ADSR parameters 605–606
 - Attack parameter 605
 - Cell envelope 604–606
 - Channel parameters 611
 - Continuous envelope 606
 - Decay parameter 606
 - Digital Sound Processor 615–638
 - FM programming 613–615
 - FM synthesis 602–614
 - Generator cell settings 608
 - High mute 607
 - Mixer 638–645
 - Multiplication factor 607
 - Mute factor 606
 - Operator cell 604
 - Operator cell settings 609–613
 - OPL chip 608
 - Percussion mode 612–613
 - Physical nature of sound 602–604
 - Principle of FM 603–604
 - Release parameter 606
 - Sampling 614–638
 - "Seeing" sound 602–604
 - Setting the channel parameters 611–612
 - Setting the channels 610–611
 - Speech output 645–648
 - Sustain parameter 606
 - Tremolo 607
 - Vibrato 607
 - Waveform 607–608
 - Speaker
 - Oscillations 323
 - Speech output 645–648
 - Sprites 160–173
 - 320x200 pixels, 256 colors 165–166
 - 320x400 pixels, 256 colors 168–170
 - 640x350 pixels, 16 colors 170–173
 - Defined 161
 - Independent movement 163–165
 - Programming 161
 - Video RAM 166–168
 - ST506 controller 343
 - Hardware configuration 343
 - Status code 340
 - Stopping program execution 437–438
 - Break flag 437
 - Critical error interrupt 437–438
 - Maintenance 437
 - Subdirectories 384–385
 - Super VGA 79, 196–206
 - Graphics modes 197–198
 - Moving access window 206
 - Reading a video mode 202–204
 - Saving/restoring setup 205
 - Setting a mode 204–205
 - Text modes 197
 - VESA compatibility 202
 - VESA standard 200–206
 - Video RAM 198–200
 - Superscalar pipelines 30
 - Synchronization 679–707
 - Between processes 705–707
 - Critical sections 688–696
 - Event objects 696–700
 - Functions 701–705
 - Interlocked variables 700–701
 - Mutexes 683–688

- Opening an existing synchronization object 706–707
 - Parallel port 262
 - Processes and threads 701
 - Race condition 680–682
 - Semaphores 691–696
 - Semaphores for controlling overflow 693–696
 - Shared use 706
 - Win32 objects/API functions 682–683
 - Synchronous communication 232–234
 - System configuration
 - Determining processor type ... 370–375
 - Determining with BIOS 367–369
 - Hardware environment (reading) 367–368
 - RAM configuration 369
 - System programming 1–62
 - BASIC examples 39–46
 - C language 53–62
 - Defined 1
 - DOS functions 1–3
 - PC hardware 3–10
 - QuickBASIC 39–46
 - ROM-BIOS 1–3
 - Three-layer model 1–3
 - Turbo Pascal 46–53
- ## T
-
- Task switching 524–525
 - The Registry
 - See *Registry*
 - Thread Local Storage (TLS)
 - See also *DLL (Dynamic Link Libraries)*
 - Threads 670–676
 - See also *Synchronization*
 - As synchronization objects 701
 - Control messages 676
 - Creating 671–672
 - Exiting 672–673
 - Functions 670
 - Multitasking 650–655
 - Priorities 673–674
 - Priority distribution 674
 - Status information 675
 - Suspension 675
 - Synchronization 674–675
 - Synchronizing with processes 679–707
 - Three-layer model 1–3
 - Time
 - Reading from BIOS (AT realtime clock) 359–361
 - Time out error 249–250
 - Time out problems 261–262
 - Time slicing 514
 - Timer (8253) 7
 - Timer frequencies 325
 - Timer interrupts 359–360
 - Tracks (Hard drives) 351–352
 - Extra tracks 354
 - Tree Views 791–826
 - Browsing 806–809
 - Communication with TreeViews 792–795
 - Drag and Drop 819–825
 - Generating a TreeView during runtime 795–802
 - Indentation of the child items 806
 - Mouse/keyboard events 817–819
 - Opening and closing 805–806
 - Owner-draw 816
 - Processing a label 811–814
 - Querying and setting items 805–806
 - Selecting the current item 809–811
 - Setting the graphics 803–805
 - Sorting 814–816
 - Tremolo 607
 - Turbo C++
 - See *C language*
 - Turbo Pascal 46–53
 - Buffers 51–53
 - Data types 46–50
 - Interrupts, calling 50–51
 - Port accesses 53
 - Typematic rate 285–287
- ## U
-
- UART registers 237–238
 - Accessing 239
 - Current status 241–243
 - Initializing 239–241
 - Interrupt mode 238
 - Polling mode 238
 - Undocumented DOS structures ... 477–480
 - DOS Info Block (DIB) 477–480
 - Drive Parameter Block (DPB) 478–480
 - Upper Memory Blocks 319–322, 421–423
 - Allocating 322
 - Releasing 322
- ## V
-
- V86 mode 535–536
 - Switching to V86 mode 535
 - VCPI 547–552
 - Client/server 547
 - Debug registers 550–551
 - EMS memory 550
 - Initializing a client-server link 548
 - Interrupt vectors 551
 - Memory management 549–550
 - Operating modes 551–552
 - Page tables 548–549
 - Status registers 550–551
 - VCPI services 548
 - Very Large Scale Integration chip 93
 - VESA
 - See *VESA standard*
 - VESA local bus 5–6
 - VESA standard 206
 - Graphics modes 201
 - Saving/restoring setup 205
 - Setting a mode 204–205
 - Super VGA compatibility 202
 - VESA BIOS functions 201–202
 - Video modes 202–204
 - VGA BIOS 158–159
 - VGA cards 115–196
 - 16 color graphics mode 146–150
 - 256 color graphics mode 150–155
 - Bitplanes 136–145
 - Changing pixel display 120–121
 - Character map structure 125–127
 - Character structure 118
 - Character table structure 123–127
 - Character wrapping 132–133
 - Color selection 155–160
 - DAC 194–196
 - Digital to Analog Converter ... 194–196
 - Disabling screen 135–136
 - Displaying 512 characters 125–127
 - Displaying special characters . 127–128
 - FGA monitors 116
 - Font loading/defining 118–120
 - Font selection/programming .. 117–129
 - Graphics mode (16 colors) 146–150
 - Graphics mode (256 colors) ... 150–155
 - Graphics mode fonts 129
 - Graphics window 127–128
 - Horizontal pel panning register 130–131, 133–134
 - Identifying 116–117
 - Monitor selection 115–117
 - Pel panning register 129–131
 - Scrolling (smoother) 129–135
 - Scrolling in text mode 134–135
 - Scrolling screen contents 131–132
 - Sprites 160–173
 - Vertical pel panning register 131
 - VGA BIOS 118–120, 158–159
 - VGA cards, Bitplanes 136–145
 - Dividing video RAM 136
 - Graphics controller 137–138
 - Latch registers 137
 - Read mode 0 138–139
 - Read mode 1 139–141
 - Write mode 0 141–143
 - Write mode 1 143
 - Write mode 2 144
 - Write mode 3 144–145
 - VGA cards, Color selection 155–160
 - DAC color table 156–157
 - Palette registers 155–156
 - Using BIOS to set color 157–158
 - VGA BIOS 158–159
 - VGA cards, Registers 173–196
 - Attribute controller 187–190
 - CRT controller registers 175–184
 - Graphics controller 190–194
 - Sequencer controller 184–187

- VGA cards, Scrolling 129–135
 Character wrapping 132–133
 Coordinating with CRT controller
 133–134
 Horizontal pel panning register
 130–131
 Horizontal scrolling 130
 Pel panning register 129–131
 Screen contents 131–132
 Text mode 134–135
 Vertical pel panning register 131
 Vertical scrolling 131
 VGA cards, Sprites 160–173, 175–200
 VGA monitors 116
 Vibrato 607
 Video BIOS 80–88
 Activating a screen page 85
 Character output 86–87
 Control functions 81–82
 Extensions 80–81
 Functions 80–88
 Initializing video mode 84
 Programming text cursor 84–85
 Reading characters 87
 Screen coordinates 83
 Screen scrolling 87–88
 Selecting background colors 82–83
 Selecting character colors 82–83
 String output 87
 Video BIOS functions 80–88
 Activating a screen page 85
 Character output 86–87
 Control functions 81–82
 Initializing video mode 84
 Programming text cursor 84–85
 Reading characters 87
 Screen coordinates 83
 Screen scrolling 87–88
 Selecting background colors 82–83
 Selecting character colors 82–83
 String output 87
 Video cards 77–206
 Address space of the PC 96
 Anatomy 91–101
 Bitplanes 136–145
 Color/Graphics Adapter (CGA)
 108–114
 CRT controller registers 93–95
 CRTC 92–93
 Determining type 88–91
 Disabling screen 135–136
 EGA cards 115–196
 Hercules Graphics Card 104–107
 History 77–80
 Operation 91–101
 Overscan 93
 Refresh rate 92
 Screen page (activating) 85
 Screen scrolling 87–88
 See also specific card/adaptor
 Structure 95–96
 Super VGA cards 196–206
 Text cursor (programming) 84–85
 VGA cards 115–196
 Video BIOS 80–88
 Video mode 84
 Video RAM 96–101
 Video cards, Determining type 88–91
 Video cards, History 77–80
 8514/A 80
 Color/Graphics Adapter 77–78
 Enhanced Graphics Adapter 78–79
 Hercules Graphics Adapter 78
 Memory Controller Gate Array .. 79–80
 Monochrome Display Adapter 77
 Super VGA 79
 Video Graphics Array 79
 Video cards, Operation 91–101
 Video cards, Structure 95–96
 Video Electronic Standards Association
 See also VESA standard 201
 Video Graphics Array (VGA) 79
 Video RAM 96–101
 Addressing 97
 Character locations 98
 Character organization 98
 Dividing into bitplanes 136
 Extended text modes 98
 Hercules Graphics Card 104–105
 Intercepting access 538
 MDA card 102
 Screen pages 99
 Scrolling screen contents EGA/VGA
 cards 131–132
 Sprites 166–168
 Structure in text mode 97–98
 Super VGA cards 198–200
 Virtual Control Programming Interface
 (VCPi)
 See *VCPI*
 Virtual memory 514–515
 Virtual Memory Management 711–755
 Access tests 726–727
 Accessing existing MMF objects
 739–740
 Accessing memory with Win32 API ...
 719–720
 Basics 711–718
 Creating memory maps 739–744
 Functions 720–737
 Mapping to the memory 740–742
 Memory mapped files (MMF) 737–742
 Page attributes 724–725
 PE (Portable Executable) format
 742–755
 Querying the memory status .. 727–737
 Unmapping a mapping 742
 Virtual functions 722
 Virtual memory management
 529–531, 533
 Virtual-86 Mode 535–536
 VLSI chip 93
 Volume label names 377
 Volumes
 CD-ROM 571
W
 Wait state 5
 Waveform 607–608
 WIN32 API
 Accessing memory 719–720
 Win32 API
 Accessing The Registry 957–971
 Functions controlling processes 655
 Functions environment variables
 processes 662–664
 Synchronization objects 682–683
 Thread functions 670
 Windows
 DOS programs (executing) 473–474
 Windows 95
 Common Controls 777–866
 Image List 778–791
 List Views 826–866
 Multitasking 649–655
 OLE (Object Linking and Embedding)
 867–888
 Processes 655–670
 Registry 951–975
 Threads 670–676
 Tree Views 791–826
 Virtual Memory Management 711–755
 Write mode 0 141–143
 Write mode 1 143
 Write mode 2 144
 Write mode 3 144–145
X
 XT BIOS variables
 Listed and explained 73–75

BIOS Interrupts And Functions

The various functions, which the ROM-BIOS makes available for the basic communication between a program and the hardware, can be accessed using interrupts 10H to 1AH. Besides functions for the access to the video-hardware, the keyboard, hard drives and diskette drives, this includes checking configuration data, as well as programming of the serial and parallel interface and the battery-buffered real-time clock.

Here is an overview of the various interrupts and their services. Please note, the various functions of Interrupt 13H are explained twice, separated according to their use in relation to diskette and hard drives. Depending on the need for access to diskettes or hard drives, please consult the proper section.

Unless otherwise stated, the various functions are available on all types of PCs. An entire series of functions have only been available since the introduction of the XT models, others only since the introduction of the AT series. The many BIOS enhancements which companies like Compaq added to the original ROM-BIOS were omitted here, since they did not establish themselves as a standard. This is also valid for the enhanced functions of the PS/2 systems of IBM, which are completely compatible with the functions presented here.

Interrupt 10H, Function 00H	BIOS
Video: Set video mode	

Selects and initializes a video mode and clears the screen. This function is a fast method of clearing the screen while maintaining the current video mode.

Input	AH = 00H
	AL = Video mode
	0: 40x25 text mode, monochrome (color card)
	1: 40x25 text mode, color (color card)
	2: 80x25 text mode, monochrome (mono card)
	3: 80x25 text mode, color (color card)
	4: 320x200 4-color graphics (color card)
	5: 320x200 4-color graphics (color card)
	(colors displayed in monochrome)
	6: 640x200 2-color graphics (color card)
	7: Internal mode (mono card)

Output	No output
---------------	-----------

Remarks:

The colors for modes 4, 5 and 6 can be set with function 11.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 01H	BIOS
Video: Define cursor type	

Defines the starting and ending lines of the cursor. This cursor exists independently of the current screen page.

Input	AH = 01H
	CH = Starting line of the cursor
	CL = Ending line of the cursor

Output	No output
---------------	-----------

Remarks

The values allowed for the cursor's starting and ending line depend on the installed video card. The following values are permitted:

Monochrome display cards:	0-13
Color display cards:	0-7

BIOS defaults to the following values:

Monochrome display cards:	11-12
Color display cards:	6-7

You can use this function to set the cursor only within the permitted ranges. Setting cursor lines outside these parameters may result in an invisible cursor or system problems.

The contents of the BX, CX, DX registers and the segment registers SS, CS and DS are not affected by this function. The contents of all the other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 02H	BIOS
Video: Position cursor	

Repositions the cursor, which determines the screen position for character output by using one of the BIOS functions.

Input	AH = 02H
	BH = Screen page number
	DH = Screen line
	DL = Screen column

Output	No output
---------------	-----------

Remarks

The blinking cursor moves through this function when the addressed screen page is the current screen page.

Values for the screen line parameter range from 0 to 24.

Values for the screen column parameter range from 0 to 79 (for an 80-column display) or from 0 to 39 (for a 40-column display), depending on the selected video mode.

You can make the cursor disappear by moving it to a nonexistent screen position (e.g., column 0, line 25).

The number of the screen page parameter depends on how many screen pages are available to the video card.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 03H**BIOS**

Video: Read cursor position

Senses the text cursor's position, starting line and ending line in a screen page.

Input

AH = 03H

BH = Screen page number

Output

DH = Screen line in which the cursor is located

DL = Screen column in which the cursor is located

CH = Starting line of the blinking cursor

CL = Ending line of the blinking cursor

Remarks

The number of the screen page parameter depends on how many screen pages are available to the video card.

Line and column coordinates are related to the text coordinate system.

The contents of the BX register and the SS, CS and DS segment registers are not affected by this function. The contents of all the other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 04H**BIOS**

Video: Read lightpen position

Senses the position of the lightpen on the screen if applicable.

Input

AH = 04H

Output

AH = 0: Lightpen position unreadable

AH = 1: Lightpen position readable

DH = Screen line of the lightpen (text mode)

DL = Screen column of the lightpen (text mode)

CH = Screen line of the lightpen (graphic mode)

BX = Screen column of the lightpen (graphic mode)

Remarks:

This function call must be repeated until 1 is returned in the AH register, because only then can coordinates be read from the other registers.

Coordinates indicated represent the current video mode's resolution.

Usually the coordinates of the light pen cannot be accurately sensed in the graphic mode. The Y-coordinate (line) is always a multiple of 2, so it isn't possible to determine whether the lightpen is in line 8 or 9. The X-coordinate (column) is always a multiple of 4 in 320x200 graphic mode and a multiple of 8 in the 640x200 bitmap mode.

The contents of the CL register and the SS, CS and DS segment registers are not affected by this function. The contents of all the other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 05H	BIOS
Video: Select current screen page	

Selects the current screen page (text mode only) which should be displayed.

Input

AH = 05H

AL = Screen page number

Output

No output

Remarks

The number of the screen page depends on the number of screen pages available to the video card.

On switching to a new screen page, the screen cursor points to the position of the text cursor in this page.

Switching between various screen pages does not affect their contents (the individual characters).

You can write characters to an inactive screen page.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of the other registers, such as the SI and DI registers, may change.

Interrupt 10H, Function 06H	BIOS
Video: Initialize window/scroll text upward	

Clears window or scrolls a portion of the current screen page up by one or more lines, depending on the input.

Input

AH = 06H

AL = Number of window lines to be scrolled upward (0=clear window)

CH = Screen line of the upper left corner of the window

CL = Screen column of the upper left corner of the window

DH = Screen line of the lower right corner of the window

DL = Screen column of the lower right corner of the window

BH = Color (attribute) for blank line(s)

Output No output

Remarks

Initializing a window (placing a 0 in the AL register) fills the window with blank spaces (ASCII code 32).

The contents of the lines scrolled out of the window are lost and cannot be restored.

Function 0 of this interrupt is better for clearing the entire screen.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 07H

BIOS

Video: Initialize window/scroll text downward

Clears window or scrolls a portion of the current screen page up by one or more lines, depending on the input.

Input AH = 07H

AL = Number of window lines to be scrolled downward (0=clear window)

CH = Screen line of the upper left corner of the window

CL = Screen column of the upper left corner of the window

DH = Screen line of the lower right corner of the window

DL = Screen column of the lower right corner of the window

BH = Color (attribute) for blank line(s)

Output No output

Remarks

This function only affects the current screen page.

Initializing a window (placing a 0 in the AL register) fills the window with blank spaces (ASCII code 32).

The contents of the lines scrolled out of the window are lost and cannot be restored.

Function 0 of this interrupt is better for clearing the entire screen.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 08H

BIOS

Video: Read character/attribute

Reads the ASCII code of the character at the current cursor position and its color (attribute).

Input	AH = 08H
	BH = Screen page number
Output	AL = ASCII code of the character
	AH = Color (attribute)

Remarks

The number of the screen page depends on the number of screen pages made available to the video card.

This function can also be called in graphic mode. The function compares the bit pattern of the character on the screen with the bit pattern of the character in character ROM of the video card and with the character patterns stored in a RAM table whose addresses appear in interrupt 1FH. If the character cannot be identified, the AL register contains the value-0 after the function call.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of the other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 09H	BIOS
Video: Write character/attribute	

Writes a character with a certain color (attribute) to the current cursor position in a predefined screen page.

Input	AH = 09H
	BH = Screen page number
	CX = Number of times to write the character
	AL = ASCII code of the character
	BL = Attribute

Output	No output
---------------	-----------

Remarks

If the character should be displayed several times (the value of the CX register is greater than 1), all characters must fit into the current screen line in the graphic mode.

The control codes (e.g., bell, carriage return) appear as normal ASCII codes.

This function can display characters in graphic mode. The patterns of the characters, with the codes from 0 to 127, are determined by a table in ROM. The patterns of the characters with the codes from 128 to 255 are determined by a RAM table that was previously installed by the DOS GRAFTABL command.

In text mode, the contents of the BL register define the attribute byte of the character. In graphic mode this register determines the color of the character. The 640x200 bitmap mode only allows the values 0 and 1 for selecting colors from the color palette. The 320x200 bitmap mode only allows the values 0 to 3 for selecting colors from the color palette.

If the graphic mode is active during character output and bit 7 of the BL register is set, an exclusive OR is performed on the character pattern and the graphic pixels behind the character pattern.

After character output, the cursor remains in the same position as the character.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 0AH	BIOS
Video: Write character	

Writes a character to the current cursor position in a predefined screen page by using the color of the character previously at this position.

Input	AH = 0AH
	BH = Screen page number
	CX = Number of times to write the character
	AL = ASCII code of the character

Output	No output
---------------	-----------

Remarks

If the character should be displayed several times (the value of the CX register is greater than 1), all characters must fit into the current screen line in the graphic mode.

The control codes (e.g., bell, carriage return) appear as normal ASCII codes.

This function can display characters in graphic mode. The patterns of the characters with the codes from 0 to 127 are determined by a table in ROM and the patterns of the characters with the codes from 128 to 255 are determined by a RAM table previously installed by the GRAFTABL command.

In text mode, the contents of the BL register define the attribute byte of the character. In graphic mode this register determines the color of the character. The 640x200 bitmap mode only allows the values 0 and 1 for selecting colors from the color palette. The 320x200 bitmap mode only allows the values 0 to 3 for selecting colors from the color palette.

If the graphic mode is active during character output and bit 7 of the BL register is set, an exclusive OR is performed on the character pattern and the graphic pixels behind the character pattern.

The cursor remains in the same position after character output.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 0BH	BIOS
Video: Select palette	

Selects the border and background color for graphic or text mode.

Input	AH = 0BH
	BH = 0
	BL = Border/background color

Output	No output
---------------	-----------

Remarks

In graphic mode, the color value passed defines the color of both the border and background. In text mode, the background color of each character is defined individually, so the passed color value only defines the color of the screen border.

Values for the color passed can range from 0 to 15.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 0BH, Subfunction 1**BIOS**

Video: Select color palette

Selects one of the two color palettes for the 320x200 bitmapped graphic mode.

Input

AH = 0BH

BH = 1

BL = Color palette number

Output

No output

Remarks

Two color palettes are available. They have the numbers 0 and 1 and contain the following colors:

Palette 0: Green, red, yellow

Palette 1: Cyan, magenta, white

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 0CH**BIOS**

Video: Write graphic pixel

Draws a color pixel at the specified coordinates in graphic mode.

Input

AH = 0CH

AL = Pixel color value (see below)

BH = Graphics page

CX = Screen column

DX = Screen line

Output

No output

Remarks

The pixel value color parameter depends on the current graphic mode. 640x200 bitmapped mode only permits the values 0 and 1. In the 320x200 bitmapped mode, the values 0 to 3 are permitted, which generates a certain color according to the chosen

color palette. 0 represents the selected background color; 1 represents the first color of the selected color palette; 2 represents the second color of the color palette, etc.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 0DH

BIOS

Video: Read graphic pixel

Reads the color value of a pixel at the specified coordinates in the current graphic mode.

Input

AH = 0DH

DX = Screen line

CX = Screen column

Output

AL = Pixel color value

Remarks

The pixel color value parameter depends on the current graphic mode. 640x200 bitmapped mode permits the values 0 and 1 only. In the 320x200 bitmapped mode, the values 0 to 3 are permitted, which generates a certain color according to the color palette chosen. 0 represents the selected background color; 1 represents the first color of the selected color palette; 2 represents the second color of the color palette, etc.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 0EH

BIOS

Video: Write character

Writes a write character at the current cursor position in the current screen page. The new character uses the color of the character that was previously in this position on the screen.

Input

AH = 0EH

AL = ASCII code of the character

BL = Foreground color of the character (graphic mode only)

Output

No output

Remarks

This function executes control codes (e.g., bell, carriage return) instead of reading them as ASCII codes. For example, the function sounds a beep instead of printing the bell character.

After this function displays a character, the cursor position increments so the next character appears at the next position on the screen. If the function reaches the last display position, the display scrolls up one line and output continues in the first column of the last screen line.

The foreground color parameter depends on the current graphic mode. 640x200 bitmapped mode only permits the values 0 and 1. In the 320x200 bitmapped mode, the values 0 to 3 are permitted, which generates a certain color according to the chosen

color palette. 0 represents the selected background color; 1 represents the first color of the selected color palette; 2 represents the second color of the color palette, etc. The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 0FH	BIOS
Video: Read display mode	

Reads the number of the current video mode, the number of characters per line and the number of the current screen page.

Input AH = 0FH

Output AL = Video mode

- 0: 40x25 text mode, monochrome (color card)
- 1: 40x25 text mode, color (color card)
- 2: 80x25 text mode, monochrome (mono card)
- 3: 80x25 text mode, color (color card)
- 4: 320x200 4-color graphics (color card)
- 5: 320x200 4-color graphics (color card)
- (colors represented in monochrome)
- 6: 640x200 2-color graphics (color card)
- 7: Internal mode (mono card)

AH = Number of characters per line

BH = Current screen page number

Remarks

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 13H	BIOS (AT only)
Video: Write character string	

Displays a character string on the screen, starting at a specified screen position on a specified screen page. The characters are taken from a buffer whose address passes to the function.

Input AH = 13H

AL = Output mode (0-3)

- 0: Attribute in BL, retain cursor position
- 1: Attribute in BL, update cursor position
- 2: Attribute in the buffer, retain cursor position
- 3: Attribute in the buffer, update cursor position

BH = Screen page number
 BL = Attribute byte of the character (modes 0 and 1 only)
 BP = Offset address of the buffer
 CX = Number of characters to be displayed
 DH = Display line
 DL = Display column
 ES = Segment address of the buffer

Output No output

Remarks

Modes 1 and 3 set the cursor position following the last character of the character string. On the next call of a BIOS function for character output, the next string of characters appears following the original character string. This does not occur in the modes 0 and 2.

In modes 0 and 1, the buffer contains only the ASCII codes of the characters to be displayed. The BL register contains the color of the character string. However, in modes 2 and 3 each character has its own attribute byte when the character is stored in the buffer. The BL register doesn't have to be loaded in this mode. Even though the character string is twice as long in these modes as the number of the characters to be displayed, the CX register requires only the number of ASCII characters in the string and not the total length of the character string.

Control codes (e.g., bell) are interpreted as control codes only, and not as characters.

When the string reaches the last position on the screen, the display scrolls upward by one line and output continues in the first column of the last screen line.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

Interrupt 11h

BIOS

Determine configuration

Reads the configuration of the system as recorded during the booting process.

Input No input

Output AX = Configuration

PC and XT

Bit 0: 1 if the system has one or more disk drives
 Bit 1: Unused

- Bits 2-3: RAM available on main circuit board
 - 00: 16K
 - 01: 32K
 - 10: 48K
 - 11: 64K
- Bits 4-5: Video mode after system boot
 - 00: Unused
 - 01: 40x25, color card
 - 02: 80x25, color card
 - 03: 80x25, mono card
- Bits 6-7: Number of disk drives in the system if bit 0 is equal to 1
 - 00: 1 disk drive
 - 01: 2 disk drives
 - 10: 3 disk drives
 - 11: 4 disk drives
- Bit 8: 0 when a DMA chip is present
- Bits 9-11: Number of RS-232 cards connected
- Bit 12: 1 when system has a joystick attached
- Bit 13: Unused
- Bits 14-15: Indicates the number of printers available

AT

- Bit 0: 1 if the system has one or more disk drives
- Bit 1: 1 when a math coprocessor exists in the system
- Bit 2-3: Unused
- Bit 4-5: Video mode during system boot
 - 00: Unused
 - 01: 40x25, color card
 - 02: 80x25, color card
 - 03: 80x25, mono card
- Bits 6-7: Number of disk drives in the system if bit 0 is equal to 1
 - 00: 1 disk drive
 - 01: 2 disk drives

- 10: 3 disk drives
- 11: 4 disk drives
- Bit 8: Unused
- Bits 9-11: Number of RS-232 cards connected
- Bit 12-13: Unused
- Bits 14-15: Indicates the number of printers available

Remarks

The type of PC must be known (PC, XT or AT) to properly interpret the meanings of the individual bits of the configuration word.

The memory size indicated in bits 2 and 3 of the PC/XT configuration word refers only to the main circuit board. Interrupt 12H lets you determine the total amount of available memory.

The video mode recorded in bits 4 and 5 is the mode that was activated when the system was switched on. To determine the current video mode use function 15 of interrupt 10H. The contents of the AX register are affected by this function.

Interrupt 12h	BIOS
Determine memory size	

Input No input

Output AX = Memory size in kilobytes

Remarks

The PC and the XT can accept a maximum of 640K of RAM. The AT accepts up to 14 megabytes of RAM memory beyond the 1 megabyte limit. The memory size returned by this function ignores this extended memory. To determine the memory size beyond the 1 megabyte limit, use function 88H of interrupt 15H (available only on the AT).

The contents of the AX register are affected by this function.

Interrupt 13H, Function 00H	BIOS
Diskette: Reset	

Resets the disk controller and any connected disk drives. A reset should be executed after each disk operation during which an error occurred.

Input AH = 00H

DL = 0 or 1

Output Carry flag=0: Operation completed (AH=0)

Carry flag=1: Error (AH=error code)

Remarks

The value in the DL register is unnecessary since all the disk drives execute a reset. XT and AT models use this register to determine whether a reset should be performed on the disk drives or the hard drive.

The following error codes can occur:

01H:	Function number not permitted
02H:	Address not found
03H:	Write attempt on write protected disk
04H:	Sector not found
08H:	DMA overflow
09H:	Data transmission beyond segment border
10H:	Read error
20H:	Error in disk controller
40H:	Track not found
80H:	Time out error, unit not responding

The contents of the BX, CX, DX, SI, DI, PB registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 01H	BIOS
Diskette: Read status	

Reads the status of the disk drive since the last disk operation.

Input

AH = 01H

DL = 0 or 1

Output

Carry flag=0: Operation completed (AH=0)

Carry flag=1: Error (AH=error code)

Remarks

The value in the DL register is unnecessary, since disk drives constantly return their status. XT and AT models use this register to determine whether the status of the hard drive should be checked.

The following error codes can occur:

01H:	Function number not permitted
02H:	Address not found
03H:	Write attempt on write protected disk
04H:	Sector not found
08H:	DMA overflow
09H:	Data transmission beyond segment border
10H:	Read error

- 20H: Error in disk controller
- 40H: Track not found
- 80H: Time out error, unit not responding

The contents of the BX, CX, DX, SI, DI, PB registers and the segment registers are not affected by this function. The contents of all other registers may change.

The following error codes can occur:

- 01H: Function number not permitted
- 02H: Address not found
- 03H: Write attempt on write protected disk
- 04H: Sector not found
- 08H: DMA overflow
- 09H: Data transmission beyond segment border
- 10H: Read error
- 20H: Error in disk controller
- 40H: Track not found
- 80H: Time out error, unit not responding

The contents of the BX, CX, DX, SI, DI, PB registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 02H

BIOS

Diskette: Read disk

Reads one or more disk sectors into a buffer.

Input

- AH = 02H
- AL = Number of sectors to be read
- BX = Offset address of buffer
- CH = Track number
- CL = Sector number
- DH = Disk side number (0 or 1)
- DL = Disk drive number
- ES = Buffer segment address

Output

- Carry flag=0: Operation completed (AH=0)
- Carry flag=1: Error (AH=error code)

Remarks

The number of sectors to be read into the AL register is limited to sectors which logically follow each other on a track on one side of the disk.

The following error codes can occur:

01H:	Function number not permitted
02H:	Address not found
03H:	Write attempt on a write protected disk
04H:	Sector not found
08H:	DMA overflow
09H:	Data transmission over segment border
10H:	Read error
20H:	Error in disk controller
40H:	Track not found
80H:	Time out error, drive not responding

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all the other registers may change.

Interrupt 13H, Function 03H	BIOS
Diskette: Write to disk	

Writes one or more sectors to a disk. The data to be transmitted are taken from a buffer.

Input	AH = 03H
	AL = Number of sectors to be written
	BX = Offset address of buffer
	CH = Track number
	CL = Sector number
	DH = Disk side number (0 or 1)
	DL = Disk drive number
	ES = Buffer segment address
Output	Carry flag=0: Operation completed (AH=0)
	Carry flag=1: Error (AH=error code)

Remarks

The number of sectors that can be written in the AL register is limited to sectors which logically follow each other on a track on one side of the disk.

The following error codes can occur:

01H:	Function number not permitted
02H:	Address not found
03H:	Write attempt on a write protected disk
04H:	Sector not found
08H:	DMA overflow
09H:	Data transmission over segment border
10H:	Read error
20H:	Error in disk controller
40H:	Track not found
80H:	Time out error, drive not responding

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 04H**BIOS**

Diskette: Verify disk sectors

Compares one or more sectors on disk with the data stored in a buffer. This can be used to verify the data was properly saved to disk.

Input

AH =	04H
AL =	Number of sectors to be verified
BX =	Offset address of buffer
CH =	Track number
CL =	Sector number
DH =	Disk side number (0 or 1)
DL =	Disk drive number
ES =	Buffer segment address

Output

Carry flag=0:	Operation completed (AH=0)
Carry flag=1:	Error (AH=error code)

Remarks

The number of sectors to be verified in the AL register is limited to sectors which logically follow each other on a track on one side of the disk.

The following error codes can occur:

01H:	Function number not permitted
02H:	Address not found
03H:	Write attempt on a write protected disk
04H:	Sector not found
08H:	DMA overflow
09H:	Data transmission over segment border
10H:	Read error
20H:	Error in disk controller
40H:	Track not found
80H:	Time out error, drive not responding

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 05H**BIOS**

Diskette: Format track

Formats a complete track on one side of a disk. A buffer which contains information about the sectors to be formatted must be passed to the function.

Input

AH =	05H
AL =	Number of sectors to be formatted
BX =	Offset address of buffer
CH =	Track number
DH =	Disk side number (0 or 1)
DL =	Disk drive number
ES =	Buffer segment address

Output

Carry flag = 0:	Operation completed (AH=0)
Carry flag = 1:	Error (AH=error code)

Remarks

The number of sectors to be formatted is limited to sectors which logically follow each other on a track on one side of the disk.

The buffer passed in ES:BX contains an entry consisting of four consecutive bytes for every sector to be formatted.

- 1: Track number
- 2: Page number
- 3: Logical sector number
- 4: Number of bytes in this sector:
- 0: 128 bytes
- 1: 256 bytes
- 2: 512 bytes (PC standard)
- 3: 1,024 bytes

The logical sector number increments continuously, but may not be the same as the physical sector number.

The following error codes can occur:

- 01H: Function number not permitted
- 02H: Address not found
- 03H: Write attempt on a write protected disk
- 04H: Sector not found
- 08H: DMA overflow
- 09H: Data transmission over segment border
- 10H: Read error
- 20H: Error in disk controller
- 40H: Track not found
- 80H: Time out error, drive not responding

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all the other registers may change.

Interrupt 13H, Function 15H	BIOS (AT only)
Diskette: Determine drive type	

Senses disk change and drive type. The AT supports both the standard 320/360K drives and the 1.2 megabyte drives.

Input

AH = 15H

DL = Disk drive number (0 or 1)

Output

Carry flag = 0: Operation completed (AH=unit type)

AH = 0: Device not present

AH=1: Unit does not recognize disk change

AH=2: Unit recognizes disk change

AH=3: Hard drive (see remarks below)

Carry flag=1: Error

Remarks

The AT has a controller which selectively controls 2 disk drives and a hard drive, or one disk drive and 2 hard drives. In the latter case, the first hard drive has the number 1 and can be accessed with this function.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 16H

BIOS (AT only)

Diskette: Media change

Senses a disk change. The AT supports both the standard 320/360K drives and the 1.2 megabyte drives. This function reads any disk change that may have occurred since the last disk access.

Input

AH = 16H

DL = Disk drive number (0 or 1)

Output

AH = 0: No disk change

AH = 6: Disk changed since last disk access

Remarks

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 17H

BIOS (AT only)

Diskette: Determine disk format

Determines the format of a disk. The AT's 1.2 megabyte disk drive can read both 320/360K disks and 1.2 megabyte disks. While the BIOS can determine disk format during a read or write access, it first must be informed of the format. Function 23 must be called on the AT before you can call function 5 (format).

Input

AH = 17H

AL = Format

AL=1: 320/360K format on 320/360K drive

AL=2: 320/360K format on 1.2 megabyte drive

AL=3: 1.2 Meg format on 1.2 megabyte drive

Output

Carry flag = 0: Operation completed

Carry flag = 1: Error

Remarks

The following error codes can occur:

01H:	Function number not permitted
02H:	Address not found
03H:	Write attempt on a write protected disk
04H:	Sector not found
08H:	DMA overflow
09H:	Data transmission over segment border
10H:	Read error
20H:	Error in disk controller
40H:	Track not found
80H:	Time out error, drive not responding

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 18H	BIOS (AT and higher)
Diskette: Determine disk format	

Determining:Disk format Determines the format of a disk. This function replaces function 17H for checking disk format in AT BIOS.

Input

AH =	18H
CH =	Highest track number
CL =	Highest sector number
DL =	Drive number (0 or 1)

Output

Carry flag = 0:	Operation completed
Carry flag = 1:	Error

Remarks

This function should be called before calling function 05H (Format track) for the first time, to configure the BIOS to the proper format.

Error-Codes; see function 00H.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 00H	BIOS (XT and AT only)
Hard drive: Reset	

Resets the hard drive controller and any interfaced hard drives. A reset should be executed after every hard drive operation during which an error was reported.

Input AH = 00H
 DL = 80H or 81H

Output Carry flag = 0: Operation completed (AH=0)
 Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

The value in the DL register is unnecessary since all the hard drives execute a reset. XT and AT models use this register to determine whether a reset should be performed on the disk drives or on the hard drive.

The following error codes can occur:

01H: Addressed function or unit not available
 02H: Address not found
 04H: Sector not found
 05H: Error on controller reset
 07H: Error during controller initialization
 09H: DMA transmission error: Segment border exceeded
 0AH: Defective sector
 10H: Read error
 11H: Read error corrected by ECC
 20H: Controller defect
 40H: Search operation failed
 80H: Time out, unit not responding
 AAH: Unit not ready
 CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 02H	BIOS (XT and AT only)
Hard drive: Read disk	

Reads the status of the hard drive since the last hard drive operation.

Input AH = 01H
 DL = 80H or 81H

Output Carry flag = 0: Operation completed (AH=0)
 Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

The value in the DL register is unnecessary since the status is consistently returned for each disk drive. XT and AT models use this register to determine whether the status of the disk drives or hard drive should be checked.

The following error codes can occur:

01H: Addressed function or unit not available
 02H: Address not found
 04H: Sector not found
 05H: Error on controller reset
 07H: Error during controller initialization
 09H: DMA transmission error: Segment border exceeded
 0AH: Defective sector
 10H: Read error
 11H: Read error corrected by ECC
 20H: Controller defect
 40H: Search operation failed
 80H: Time out, unit not responding
 AAH: Unit not ready
 CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of the other registers may change.

Interrupt 13H, Function 01H	BIOS (XT and AT only)
Hard drive: Read status	

Reads one or more hard drive sectors into a buffer.

Input	AH = 02H
	AL = Number of sectors to be read (1-128)
	BX = Offset address of buffer
	CH = Cylinder number
	CL = Sector number
	DH = Read/write head number
	DL = Hard drive number (80H or 81H)
	ES = Buffer segment address
Output	Carry flag=0: Operation completed (AH=0)
	Carry flag=1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H. Since the eight bits of the CH register can address only 256 cylinders at a time, bits 6 and 7 of the CL register (sector number) form bits 8 and 9 of the cylinder number, which enables the addressing of up to 1,023 cylinders at a time.

If several sectors are being read and the system reaches the last sector of a cylinder, reading continues at the first sector of the next cylinder of the next read/write head. If the system reaches the last read/write head, reading continues on the first sector of the following cylinder on the first read/write head.

The following error codes can occur:

01H:	Addressed function or unit not available
02H:	Address not found
04H:	Sector not found
05H:	Error on controller reset
07H:	Error during controller initialization
09H:	DMA transmission error: Segment border exceeded
0AH:	Defective sector
10H:	Read error
11H:	Read error corrected by ECC
20H:	Controller defect
40H:	Search operation failed
80H:	Time out, unit not responding
AAH:	Unit not ready
CCH:	Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 02H	BIOS (XT and AT only)
Hard drive: Write to disk	

Writes one or more sectors to the hard drive. The data to be transmitted are taken from a buffer in the calling program.

Input

AH = 03H
 AL = Number of sectors to be written (1-128)
 BX = Offset address of buffer
 CH = Cylinder number
 CL = Sector number
 DH = Read/write head number
 DL = Hard drive number (80H or 81H)
 ES = Buffer segment address

Output

Carry flag=0: Operation completed (AH=0)
 Carry flag=1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

Since the eight bits of the CH register can address only 256 cylinders at a time, bits 6 and 7 of the CL register (sector number) form bits 8 and 9 of the cylinder number, enabling the addressing of up to 1,023 cylinders at a time.

If several sectors are being written and the system reaches the last sector of a cylinder, writing continues at the first sector of the next cylinder of the next read/write head. If the system reaches the last read/write head, writing continues on the first sector of the following cylinder on the first read/write head.

The following error codes can occur:

01H: Addressed function or unit not available
 02H: Address not found
 04H: Sector not found
 05H: Error on controller reset
 07H: Error during controller initialization
 09H: DMA transmission error: Segment border exceeded
 0AH: Defective sector
 10H: Read error
 11H: Read error corrected by ECC

20H:	Controller defect
40H:	Search operation failed
80H:	Time out, unit not responding
AAH:	Unit not ready
CCH:	Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 04H	BIOS (XT and AT only)
Hard drive: Verify disk sectors	

Verifies one or more sectors of a hard drive. Unlike the corresponding floppy disk function, the data on the hard drive are not compared with the data in memory. During data storage, four check bytes are stored for every sector; these check bytes verify the contents of a sector.

Input	AH = 04H
	AL = Number of sectors to be verified (1-128)
	BX = Offset address of buffer
	CH = Cylinder number
	CL = Sector number
	DH = Read/write head number
	DL = Hard drive number (80H or 81H)
	ES = Buffer segment address

Output	Carry flag=0: Operation completed (AH=0)
	Carry flag=1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

Since the eight bits of the CH register can only address 256 cylinders at a time, bits 6 and 7 of the CL register (sector number) form bits 8 and 9 of the cylinder number, which enables the addressing of up to 1,023 cylinders at a time.

If several sectors are being verified and the system reaches the last sector of a cylinder, verification continues at the first sector of the next cylinder of the next read/write head. If the system reaches the last read/write head, verification continues on the first sector of the following cylinder on the first read/write head.

The following error codes can occur:

01H:	Addressed function or unit not available
02H:	Address not found
04H:	Sector not found

05H:	Error on controller reset
07H:	Error during controller initialization
09H:	DMA transmission error: Segment border exceeded
0AH:	Defective sector
10H:	Read error
11H:	Read error corrected by ECC
20H:	Controller defect
40H:	Search operation failed
80H:	Time out, unit not responding
AAH:	Unit not ready
CCH:	Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 05H**BIOS (XT and AT only)**

Hard drive: Format cylinder

Formats a complete cylinder (17 sectors) of a hard drive. A buffer, which contains information about the sectors to be formatted, must be passed to the function.

Input

AH =	05H
AL =	17
BX =	Offset address of buffer
CH =	Cylinder number
CL =	1
DH =	Read/write head number
DL =	Hard drive number (80H or 81H)
ES =	Buffer segment address

Output

Carry flag=0:	Operation completed (AH=0)
Carry flag=1:	Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

Since the eight bits of the CH register can only address 256 cylinders at a time, bits 6 and 7 of the CL register (sector number) form bits 8 and 9 of the cylinder number, which enables the addressing of up to 1,023 cylinders at a time.

Since a complete cylinder is always formatted, the first sector to be formatted in the CL register is always sector 1. For the same reason the number of sectors to be formatted in the AL register is always 17, since the average hard drive operates with 17 sectors per cylinder.

The buffer, whose address is passed in ES:BX, must always be at least 512 bytes long. Only the first 34 bytes of this buffer are used for formatting the 17 sectors of a cylinder. Two succeeding bytes contain information about the corresponding physical sector. Before the function call, the first byte isn't significant. After the function call the first byte indicates whether the sector could be formatted (00H) or (80H). The second byte returns the logical sector number of the physical sector and must be placed in the buffer by calling the program before the function call.

The following error codes can occur:

01H:	Addressed function or unit not available
02H:	Address not found
04H:	Sector not found
05H:	Error on controller reset
07H:	Error during controller initialization
09H:	DMA transmission error: Segment border exceeded
0AH:	Defective sector
10H:	Read error
11H:	Read error corrected by ECC
20H:	Controller defect
40H:	Search operation failed
80H:	Time out, unit not responding
AAH:	Unit not ready
CCH:	Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 08H

BIOS (XT and AT only)

Check format

Conveys the formatting information found on the hard drive.

Input

AH =	08H
CH =	Cylinder number
CL =	Sector number
DH =	Read/write head number (0=first head)
DL =	Hard drive number

Output Carry flag=0: Operation completed (AH=0)

Carry flag=1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

Since the eight bits of the CH register can address only 256 cylinders at a time, bits 6 and 7 of the CL register (sector number) form bits 8 and 9 of the cylinder number, enabling the addressing of up to 1,023 cylinders at a time.

The total capacity of the hard drive unit in bytes can be calculated with the following formula:

Capacity = Heads * Cylinders * Sectors * 512

The following error codes can occur:

01H:	Addressed function or unit not available
02H:	Address not found
04H:	Sector not found
05H:	Error on controller reset
07H:	Error during controller initialization
09H:	DMA transmission error: Segment border exceeded
0AH:	Defective sector
10H:	Read error
11H:	Read error corrected by ECC
20H:	Controller defect
40H:	Search operation failed
80H:	Time out, unit not responding
AAH:	Unit not ready
CCH:	Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 09H

BIOS (XT and AT only)

Hard drive: Adapt to foreign drives

Interfaces other hard drives for access through BIOS functions.

Input AH = 09H

DL = Number of hard drive to be interfaced (80H or 81H)

Output Carry flag = 0: Operation completed (AH=0)

Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

BIOS takes the information about the hard drive to be interfaced (number of units, read/write heads, etc.) from a table. The address of this table for the hard drive unit numbered 80H is stored in interrupt vector 41H, and the unit numbered 81H is stored in interrupt 46H.

The following error codes can occur:

- 01H: Addressed function or unit not available
- 02H: Address not found
- 04H: Sector not found
- 05H: Error on controller reset
- 07H: Error during controller initialization
- 09H: DMA transmission error: Segment border exceeded
- 0AH: Defective sector
- 10H: Read error
- 11H: Read error corrected by ECC
- 20H: Controller defect
- 40H: Search operation failed
- 80H: Time out, unit not responding
- AAH: Unit not ready
- CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 0AH

BIOS (XT and AT only)

Hard drive: Extended read

Reads one or more sectors from the hard drive into a buffer. Besides the actual 512 bytes stored in the sector, the function also reads the four check bytes (ECC).

Input AH = 0AH

AL = Number of sectors to be read (1-127)

BX = Offset address of buffer

CH = Cylinder number

CL = Sector number
DH = Read/write head number
DL = Hard drive number (80H or 81H)
ES = Buffer segment address

Output Carry flag = 0: Operation completed (AH=0)

Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

Normally the controller computes the four check bytes. Here the buffer reads the information direct.

Since the eight bits of the CH register can only address 256 cylinders at a time, bits 6 and 7 of the CL register (sector number) form bits 8 and 9 of the cylinder number, enabling the addressing of up to 1,023 cylinders at a time.

If several sectors are being read and the system reaches the last sector of a cylinder, reading continues at the first sector of the next cylinder of the next read/write head. If the system reaches the last read/write head, reading continues on the first sector of the following cylinder on the first read/write head.

01H: Addressed function or unit not available
02H: Address not found
04H: Sector not found
05H: Error on controller reset
07H: Error during controller initialization
09H: DMA transmission error: Segment border exceeded
0AH: Defective sector
10H: Read error
11H: Read error corrected by ECC
20H: Controller defect
40H: Search operation failed
80H: Time out, unit not responding
AAH: Unit not ready
CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 0BH	BIOS (XT and AT only)
Hard drive: Extended write	

Writes one or more sectors to the hard drive. Besides the actual 512 bytes stored in a sector, four check bytes (ECC) stored at the end of every sector are transmitted from the buffer.

Input

AH = 0BH

AL = Number of sectors to be read (1-127)

BX = Offset address of buffer

CH = Cylinder number

CL = Sector number

DH = Read/write head number

DL = Hard drive number (80H or 81H)

ES = Buffer segment address

Output

Carry flag = 0: Operation completed (AH=0)

Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

Normally the controller calculates the four check bytes. Here the system reads them direct from the buffer.

Since the eight bits of the CH register can only address 256 cylinders at a time, bits 6 and 7 of the CL register (sector number) form bits 8 and 9 of the cylinder number, enabling the addressing of up to 1,023 cylinders at a time.

If several sectors are being written and the system reaches the last sector of a cylinder, writing continues at the first sector of the next cylinder of the next read/write head. If the system reaches the last read/write head, writing continues on the first sector of the following cylinder on the first read/write head.

The following error codes can occur:

01H: Addressed function or unit not available

02H: Address not found

04H: Sector not found

05H: Error on controller reset

07H: Error during controller initialization

09H: DMA transmission error: Segment border exceeded

0AH: Defective sector

10H: Read error

11H: Read error corrected by ECC

20H: Controller defect
 40H: Search operation failed
 80H: Time out, unit not responding
 AAH: Unit not ready
 CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 0CH	BIOS (XT and above)
Hard drive: Move read/write head	

Positions a hard drive read/write head over a specific cylinder. This is useful for moving the head away from a cylinder containing data, as might be done when a computer must be transported.

Input

AH = 0CH
 DL = Hard drive number (80H or 81H)
 DH = Read/write head number
 CH = Cylinder number
 CL = Sector number

Output

Carry flag=0: Operation completed
 Carry flag=1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, and the second the number 81H.

Since the 8 bits of the CH register can address only 256 cylinders, bits 6 and 7 of the sector number (CL register) form bits 8 and 9 of the cylinder number. This permits up to 1,023 cylinders to be addressed.

The read/write head is positioned automatically for these operations, so this function is not needed for normal operations.

The following error codes can occur:

01H: Addressed function or unit not available
 02H: Address not found
 04H: Sector not found
 05H: Error on controller reset
 07H: Error during controller initialization
 09H: DMA transmission error: Segment border exceeded
 0AH: Defective sector

10H: Read error
 11H: Read error corrected by ECC
 20H: Controller defect
 40H: Search operation failed
 80H: Time out, unit not responding
 AAH: Unit not ready
 CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 0DH	BIOS (XT and AT only)
Hard drive: Reset	

Resets the hard drive controller and any interfaced hard drives. A reset should be executed after every hard drive operation during which an error was reported.

Input	AH = 0DH
	DL = Hard drive drive number (80H or 81H)
Output	Carry flag=0: Operation completed (AH=0)
	Carry flag=1: Error (AH=error code)

Remarks

The value in the DL register is unnecessary since all the hard drives execute a reset. XT and AT models use this register to determine whether a reset should be performed on the disk drives or on the hard drive.

This function is identical to function 0 listed above.

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

The following error codes can occur:

01H: Addressed function or unit not available
 02H: Address not found
 04H: Sector not found
 05H: Error on controller reset
 07H: Error during controller initialization
 09H: DMA transmission error: Segment border exceeded
 0AH: Defective sector
 20H: Controller defect
 40H: Search operation failed

80H: Time out, unit not responding

AAH: Unit not ready

CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 0EH	BIOS (PS/2)
Hard drive: Read from test buffer	

Performs a data transmission test between controller and CPU in the PS/2.

Interrupt 13H, Function 0FH	BIOS (PS/2)
Hard drive: Write to test buffer	

Performs a data transmission test between controller and CPU in the PS/2.

Interrupt 13H, Function 12H	BIOS (PS/2)
Hard drive: Controller RAM diagnostic	

Performs an internal controller diagnostic in the PS/2.

Interrupt 13H, Function 13H	BIOS (PS/2)
Hard drive: Drive diagnostic	

Performs an internal drive/controller diagnostic in the PS/2.

Interrupt 13H, Function 10H	BIOS (XT and AT only)
Hard drive: Drive ready?	

Determines if the drive is ready (i.e., the last operation has been completed and the drive can perform the next task).

Input AH = 10H

DL = Hard drive drive number (80H or 81H)

Output Carry flag = 0: Drive ready (AH=0)

Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

The following error codes can occur:

01H: Addressed function or unit not available

02H: Address not found

04H: Sector not found

05H: Error on controller reset
 07H: Error during controller initialization
 09H: DMA transmission error: Segment border exceeded
 0AH: Defective sector
 10H: Read error
 11H: Read error corrected by ECC
 20H: Controller defect
 40H: Search operation failed
 80H: Time out, unit not responding
 AAH: Unit not ready
 CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 11H	BIOS (XT and AT only)
Hard drive: Recalibrate drive	

Recalibrates hard drive after an error occurs, especially after a read or write error.

Input AH = 11H
 DL = Hard drive drive number (80H or 81H)

Output Carry flag = 0: Operation completed (AH=0)
 Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

The following error codes can occur:

01H: Addressed function or unit not available
 02H: Address not found
 04H: Sector not found
 05H: Error on controller reset
 07H: Error during controller initialization
 09H: DMA transmission error: Segment border exceeded
 0AH: Defective sector
 10H: Read error

11H: Read error corrected by ECC

20H: Controller defect

40H: Search operation failed

80H: Time out, unit not responding

AAH: Unit not ready

CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 14H	BIOS (XT and AT only)
Hard drive: Controller diagnostic	

Initializes an internal diagnostic test of the hard drive controller.

Input

AH = 14H

DL = Hard drive drive number (80H or 81H)

Output

Carry flag=0: Operation completed (AH=0)

Carry flag=1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H. The following error codes can occur:

01H: Addressed function or unit not available

02H: Address not found

04H: Sector not found

05H: Error on controller reset

07H: Error during controller initialization

09H: DMA transmission error: Segment border exceeded

0AH: Defective sector

10H: Read error

11H: Read error corrected by ECC

20H: Controller defect

40H: Search operation failed

80H: Time out, unit not responding

AAH: Unit not ready

CCH: Write error

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 13H, Function 15H	BIOS (AT only)
Hard drive: Determine drive type	

Determines whether the computer hardware assigned numbers 80H and 81H are hard drives. The AT contains a controller capable of controlling both hard drives and disk drives. This controller can manage either two disk drives and one hard drive, or one disk drive and two hard drives.

Input AH = 15H
 DL = Hard drive drive number (80H or 81H)

Output Carry flag = 0: Operation completed (AH=drive type)

- 0: Equipment not available
- 1: Drive does not recognize disk change
- 2: Drive recognizes disk change
- 3: Hard drive unit

 Carry flag = 1: Error (AH=error code)

Remarks

The first hard drive is assigned the number 80H, the second is assigned the number 81H.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 14H, Function 00H	BIOS
Serial port: Initialize	

Initializes and configures a serial port. This configuration includes parameters for word length, baud rate, parity and stop bits.

Input AH = 00H
 DX = Number of serial port (0=first serial port, 1=second serial port)
 AL = Configuration parameters

 Bits 0-1: Word length

- 10(b) = 7 bits
- 11(b) = 8 bits

 Bit 2: Number of stop bits

- 00(b) = 1 stop bit
- 01(b) = 2 stop bits

Bits 3-4: Parity

00(b) = none

01(b) = odd

11(b) = even

Bits 5-7: Baud rate

000(b) = 110 baud

001(b) = 150 baud

010(b) = 300 baud

011(b) = 600 baud

100(b) = 1200 baud

101(b) = 2400 baud

110(b) = 4800 baud

111(b) = 9600 baud

Output

AH = Serial port status

Bit 0: Data ready

Bit 1: Overrun error

Bit 2: Parity error

Bit 3: Framing error

Bit 4: Break discovered

Bit 5: Transmission hold register empty

Bit 6: Transmission shift register empty

Bit 7: Time out

AL = Modem status

Bit 0: Modem ready to send status change

Bit 1: Modem on status change

Bit 2: Telephone ringing status change

Bit 3: Connection to receiver status change

Bit 4: Modem ready to send

Bit 5: Modem on

Bit 6: Telephone ringing

Bit 7: Connection to receiver modem

Remarks

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all the other registers may change.

Interrupt 14H, Function 01H	BIOS
Serial port: Send character	

Sends a character to the serial port.

Input

AH = 01H

DX = Number of serial port (0=first serial port, 1=second serial port)

AL = Character code to be sent

Output

AH: Bit 7 = 0: Character transmitted

Bit 7 = 1: Error

Bit 0-6: Serial port status

Bit 0: Data ready

Bit 1: Overrun error

Bit 2: Parity error

Bit 3: Framing error

Bit 4: Break discovered

Bit 5: Transmission hold register empty

Bit 6: Transmission shift register empty

Remarks

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 14H, Function 02H	BIOS
Serial port: Read character	

Receives a character from the serial port.

Input

AH = 02H

DX = Number of serial port (0=first serial port, 1=second serial port)

Output

AH: Bit 7 = 0: Character received:

AL = Character received

Bit 7 = 1: Error:

Bit 0-6: Serial port status:

- Bit 0: Data ready
- Bit 1: Overrun error
- Bit 2: Parity error
- Bit 3: Framing error
- Bit 4: Break discovered
- Bit 5: Transmission hold register empty
- Bit 6: Transmission shift register empty

Remarks

This function should only be called if function 3 has determined that a character is ready for reception.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 14H, Function 03H**BIOS**

Serial port: Read status

Reads the status of the serial port.

Input

AH = 03H

DX = Number of the serial port (the first serial port has the number 0)

Output

AH = Serial port status

Bit 0: Data ready

Bit 1: Overrun error

Bit 2: Parity error

Bit 3: Framing error

Bit 4: Break discovered

Bit 5: Transmission hold register empty

Bit 6: Transmission shift register empty

AL = Modem status:

Bit 0: Modem ready to send status change

Bit 1: Modem on status change

Bit 2: Telephone ringing status change

Bit 3: Connection to receiver status change

Bit 4: Modem ready to send

- Bit 5: Modem on
- Bit 6: Telephone ringing
- Bit 7: Connection to receiver modem

Remarks

This function should always be called before calling function 2 (read character).

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 15H, Function 83H	BIOS (AT only)
Cassette interrupt: Set flag after time interval	

Sets bit 7 of a flag to 1 after a certain amount of time in microseconds elapses.

- Input**
- AH = 83H
 - ES = Segment address of the flag
 - BX = Offset address of the flag
 - CX = High word of elapsed time in microseconds
 - DX = Low word of elapsed time in microseconds

Output No output

Remarks

A microsecond is a millionth of a second.

The contents of the BX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 15H, Function 84H, Subfunction 0	BIOS (AT only)
Cassette interrupt: Read joystick switch settings	

Reads the status of switches on joysticks interfaced to a PC, if game ports and joysticks are present.

- Input**
- AH = 84H
 - DX = 0

Output

- Carry flag = 1: No game port connected
- Carry flag = 0: Game port present:
- AL = Switch settings:

Bit 7=1: First joystick's first switch enabled

Bit 6=1: First joystick's second switch enabled

Bit 5=1: Second joystick's first switch enabled

Bit 4=1: Second joystick's second switch enabled

Remarks

Subfunction 1 reads the joystick position(s).

The contents of the BX, CX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 15H, Function 84H, Subfunction 1	BIOS (AT only)
Cassette interrupt: Read joystick position	

Reads the positions of joysticks interfaced to a PC if game ports and joysticks are present.

Input AH = 84H

DX = 1

Output Carry flag = 1: No game port connected

Carry flag = 0: Game port present:

AX = X-position of first joystick

BX = Y-position of first joystick

CX = X-position of second joystick

DX = Y-position of second joystick

Remarks

Subfunction 0 reads the joystick switch status.

The contents of the SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 15H, Function 85H	BIOS (AT only)
Cassette interrupt: <u>␣</u> key activated	

Responds to pressing or releasing of the ␣ key. The keyboard routine calls this function.

Input AH = 85H

AL = 0: ␣ key depressed

AL = 1: ␣ key released

Output No output

Remarks

This function acts as an intermediary for application programs, so the application program will respond appropriately when the user presses the ␣ key.

Interrupt 15H, Function 86H	BIOS (AT only)
Cassette interrupt: Wait	

Returns control to the calling program after a certain amount of time has elapsed.

Input

AH = 86H

CX = High word of pause time in microseconds

DX = Low word of pause time in microseconds

Output No output

Remarks

A microsecond is a millionth of a second.

The contents of the BX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 15H, Function 87H	BIOS (AT only)
Cassette interrupt: Move memory areas	

Moves areas of RAM from below the 1 megabyte limit to the range above the 1 megabyte limit, and from above the 1 megabyte limit to below the 1 megabyte limit.

Input

AH = 87H

CX = Number of words to move

ES = Segment address of global descriptor table

SI = Offset address of global descriptor table

Output

Carry flag = 0: No error

Carry flag = 1: Error:

AH=1: RAM parity error

AH=2: Incorrect GDT on function call

AH=3: Protected mode could not be initialized

Remarks

Only words can be transferred; individual bytes cannot be transferred.

Maximum amount of memory allowed in a transfer is 64K. The value in the CX register cannot exceed 8000H.

All interrupts are disabled during the memory block move.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 15H, Function 88h	BIOS (AT only)
Cassette interrupt: Determine memory size beyond 1 megabyte	

Determines the amount of memory installed beyond the 1 megabyte limit.

Input AH = 88H

Output AX = Memory size

Remarks

The value in the AX register represents memory in kilobytes (K).

Memory size below the 1 megabyte limit can be determined using interrupt 12H.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 15H, Function 89H	BIOS (AT only)
Cassette interrupt: Switch to virtual mode	

Switches the 80286 processor to virtual mode.

Input AH = 89H

Output No output

Remarks

This function should be called only if you know how virtual mode operates. Improper use of this function can easily lead to a system crash.

Interrupt 16H, Function 00H	BIOS
Keyboard: Read character	

Reads a character from the keyboard buffer. If the buffer doesn't contain a character, the function waits until a character is entered. Then the character is read and removed from the keyboard buffer.

Input AH = 00H

Output AL = 0: Extended key code

 AH = Extended key code

 AL>1: Normal key activated

 AL = ASCII code of key

 AH = Scan code of key

Remarks

ASCII code definition occurs independent of the keyboard. Scan codes apply only to the type of keyboard attached to the PC.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 16H, Function 01H	BIOS
Keyboard: Read keyboard for character	

Reads the keyboard buffer for a character ready to be entered. If a character is available, the function passes the character to the calling function. The character remains in the keyboard buffer and can be re-read when a program calls either function 0 (see above) or function 1. The function returns to the calling program immediately after the call.

Input AH = 01H

Output Zero flag = 1: No character in the keyboard buffer

Zero flag = 0: Character available in keyboard buffer:

AL = 0: Extended key code
AH = Extended key code

AL > 1: Normal key

AL = ASCII code of the key

AH = Scan code of the key

Remarks

ASCII code definition occurs independent of the keyboard. Scan codes only apply to the type of keyboard attached to the PC.

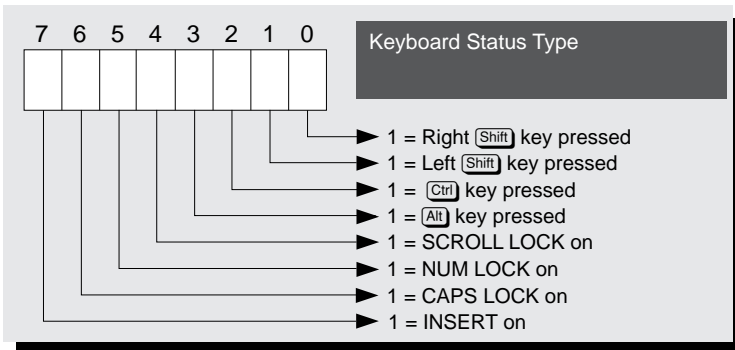
The contents of the CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 16H, Function 02H	BIOS
Keyboard: Read	

Reads and returns the status of certain control keys and various keyboard modes.

Input AH = 02H

Output AL = Keyboard status

*Remarks*

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 16H, Function 03H**BIOS (AT only)**

Keyboard: Set typematic and delay rate

Sets typematic rate and delay rate for the keyboard.

Input

AH = 03H

AL = 05H

BH = Delay before typematic

L = Typematic (repetition) rate

Output

No output

Remarks

Not every BIOS supports this function.

The following values specify the delay rate in the BH register:

Code	Delay rate
00H	1/4 second
01H	1/2 second
10H	1/4 second
11H	1 second

The following values specify the typematic rate in the BL register:

Code	RPS*	Code	RPS*	Code	RPS*	Code	RPS*
1FH	2.0	17H	4.0	0FH	8.0	07H	16.0
1EH	2.1	16H	4.3	0EH	8.6	06H	17.1
1DH	2.3	15H	4.6	0DH	9.2	05H	18.5
1CH	2.5	14H	5.0	0CH	10.0	04H	20.0
1BH	2.7	13H	5.5	0BH	10.9	03H	21.8
1AH	3.0	12H	6.0	0AH	12.0	02H	24.0
19H	3.3	11H	6.7	09H	13.3	01H	26.7
18H	3.7	10H	7.5	08H	15.0	00H	30.0
* Repetitions Per Second							

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 16H, Function 05H
BIOS (AT and above)

Keyboard: Simulate keypress

Simulates a keypress by placing the specified key code at the end of the keyboard buffer.

Input

AH = 05H

CH = Scan code of key

CL = ASCII code of key

Output

AL = 00H: O.K.

AL = 01H: Keyboard buffer full, error

Remarks

Not every BIOS supports this function.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 16H, Function 10H
BIOS (AT and above)

Keyboard: Read key on extended keyboard

Reads the keyboard buffer. This function is similar to function 00H, except that function 10H is intended for extended keyboards with 101 or 102 keys (i.e., keyboards including the **F11** and **F12** keys).

Input

AH = 10H

Output

AH = Scan code of key

AL = ASCII code of key

Remarks

Not every BIOS supports this function.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 16H, Function 11H	BIOS (since AT)
Keyboard: Read extended keyboard for character	

Reads the keyboard buffer for a character ready to be entered. This function is similar to function 01H, except that function 11H is intended for extended keyboards with 101 or 102 keys (i.e., keyboards including the **F11** and **F12** keys).

Input

AH = 11H

Output

Zero flag=1: No character in the keyboard buffer

Zero flag=0: Character available

AL = 0: Extended key code

AH=Extended key code

AL > 0: Normal key activated

AL=ASCII code of key

AH=Scan code of key

Remarks

Not every BIOS supports this function.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 17H, Function 00H	BIOS
Keyboard: Write character	

Writes a character to one of the printers interfaced to the PC.

Input

AH = 00H

AL = Character code to be printed

DX = Printer number

Output

AH = Printer status:

Bit 0=1: Time out error

Bit 1: Unused

Bit 2: Unused

Bit 3=1: Transfer error

Bit 4=0: Printer offline

Bit 4=1: Printer online

Bit 5=1: Printer out of paper

Bit 6=1: Receive mode selected

Bit 7=0: Printer busy

Remarks

Parallel port LPT1 is assigned the number 0, parallel port LPT2 is assigned the number 1 and parallel port LPT3 is assigned the number 2.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 17H, Function 01H

BIOS

Printer: Initialize printer

Printer: Initialize printer

Initializes the printer interfaced to the PC. This function should be executed before executing function 0 (see above).

Input

AH = 01H

DX = Printer number

Output

AH = Printer status

Remarks

Parallel port LPT1 is assigned the number 0, parallel port LPT2 is assigned the number 1 and parallel port LPT3 is assigned the number 2.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 17H, Function 02H

BIOS

Printer: Read printer status

Returns the status of the printer interfaced to the PC.

Input

AH = 02H

DX = Printer number

Output

AH = Printer status

Remarks

Parallel port LPT1 is assigned the number 0, parallel port LPT2 is assigned the number 1 and parallel port LPT3 is assigned the number 2. The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 18H	BIOS
Call ROM BASIC	

Accesses BASIC in ROM if a system disk cannot be found during the system bootstrap process.

Input No input

Output No output

Remarks

Very few PCs or compatibles have built-in ROM BASIC (this is a throwback from the early days of the PC). If a PC doesn't have ROM BASIC, interrupt 18H returns the system to the calling program. However, if the PC does have ROM BASIC, interrupt 18H calls BASIC. In most cases, the only way to return to DOS is by warm-starting the computer (pressing the **Ctrl** **Alt** **Del** keys) or turning the computer off and on again. Some versions of ROM BASIC allow an exit to DOS by entering the SYSTEM command from BASIC.

Interrupt 19H	BIOS
Boot process	

Boots the computer.

Input No input

Output No output

Remarks

Pressing the **Ctrl** **Alt** **Del** keys invokes this interrupt from the keyboard.

Interrupt 1AH, Function 00H	BIOS
Date and time: Read clock count	

Reads the current clock count. The clock count increments 18.2 times per second. This calculates the time elapsed since the computer was switched on.

Input AH = 00H

Output CX = High word of the clock count

DX = Low word of the clock count

AL = 0: Less than 24 hours have elapsed since the last reading

AL > 0: More than 24 hours have elapsed since the last reading

Remarks

The AT, which has a battery powered realtime clock, sets the clock count to the current time when the computer boots. PCs (which don't have realtime clocks) set the counter to 0 during booting.

The contents of the BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1AH, Function 01H	BIOS
Date and time: Set clock count	

Sets the contents of the current clock count, which increments 18.2 times per second. This calculates the time elapsed since the computer was switched on and sets the current time through this function.

Input AH = 01H

CX = High word of clock count

DX = Low word of clock count

Output No output

Remarks

The AT, which has a battery powered realtime clock, sets the clock count to the current time when the computer boots. PCs (which don't have realtime clocks) set the counter to 0 during booting. PC owners should use this function to set the current time.

The contents of the AX, BX, CX, DX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1AH, Function 02H	BIOS (AT only)
Date and time: Read realtime clock	

Reads the time from the realtime clock.

Input AH = 02H

Output Carry flag = 0: O.K.:

CH = Hours

CL = Minutes

DH = Seconds

Carry flag = 1: Dead clock battery

Remarks

All time readings appear in BCD format.

The contents of the BX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1AH, Function 03H	BIOS (AT only)
Date and time: Set realtime clock	

Sets the time on the realtime clock.

Input

AH = 03H

CH = Hours

CL = Minutes

DH = Seconds

DL = 1: Daylight Saving Time

DL = 0: Standard Time

Output No output

Remarks

All time settings must be in BCD format.

The contents of the BX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1AH, Function 04H

BIOS (AT only)

Date and time: Read date from realtime clock

Reads the current date from the realtime clock.

Input AH = 04H

Output Carry flag = 0: O.K.:

CH = Century (19 or 20)

CL = Year

DH = Month

DL = Day

Carry flag = 1: Dead clock battery

Remarks

All date readings appear in BCD format.

The contents of the BX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1AH, Function 05H

BIOS (AT only)

Date and time: Set date in realtime clock

Sets the current date in the realtime clock.

Input AH = 05H

CH = Century (19 or 20)

CL = Year
DH = Month
DL = Day

Output No output

Remarks

All date settings must be in BCD format.

The contents of the BX, CX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1AH, Function 06H

BIOS (AT only)

Date and time: Set alarm time

Sets alarm time for the current day. The alarm time triggers interrupt 4AH.

Input AH = 06H
CH = Hours
CL = Minutes
DH = Seconds

Output Carry flag=0: O.K.
Carry flag=1: Dead clock battery or programmed alarm time

Remarks

All alarm settings must be in BCD format.

During booting, interrupt 4AH points to an IRET command. If this interrupt doesn't point to a particular routine responding to the alarm, nothing will happen once the alarm time is reached.

Only one alarm time can be active at a time. If another alarm setting already exists, you must first delete it by using interrupt 26-1AH, function 7 (see below).

The contents of the BX, CX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1AH, Function 07H

BIOS (AT only)

Date and time: Reset alarm time

Clears an existing alarm setting created by using function 06H above.

Input AH = 07H

Output No output

Remarks

This function must be called when you want to change an alarm setting. Reset the alarm, then use function 06H to set the new alarm time.

The contents of the BX, CX, SI, DI, BP registers and the segment registers are not affected by this function. The contents of all other registers may change.

Interrupt 1BH**BIOS/DOS**Keyboard: **Break** key pressed

Records the occurrence of a **Ctrl Break** key combination and triggers interrupt 1BH. During the system boot, BIOS sets interrupt 1BH to an IRET command to prevent any reaction.

This routine sets a flag to indicate the user has pressed **Ctrl Break**. Following the execution of one of the DOS functions, this flag is tested for character input or output. If the system encounters **Ctrl Break**, the current program stops. In addition, when a batch file is in process, the program asks whether the batch file should be continued or terminated.

Pressing **Ctrl C** doesn't activate the interrupt. This key combination forces DOS to end the currently executing program. However, the DOS functions for character input/output search for this key combination.

To prevent termination of an application program, this interrupt can also be pointed to a user routine by pressing **Break** or **Ctrl Break**.

Input No input**Output** No output*Remarks*

Before returning control to the calling program, this interrupt must restore all registers to their previous values.

Interrupt 1CH**BIOS**

Periodic interrupt

The timer IC calls interrupt 8H approximately 18.2 times per second. After ending its task, it calls interrupt 1CH to allow an application program access to the signals from the timer IC. During booting, BIOS initializes the interrupt vector of interrupt 1CH so it points to an IRET command, which prevents any response if the interrupt is called. For example, this interrupt can be pointed to a user routine to create a constant display clock on the screen.

Input No input**Output** No output*Remarks*

This interrupt must restore all registers to their previous values before returning control to the calling program.

Interrupt 1DH**BIOS**

Video table

Sets a pointer to a table. The vector of this interrupt in the vector table, starting at address 0000:0074, stores the offset and segment address of this table. The table itself contains a collection of parameters used by BIOS for initializing a certain video

mode. This involves the 16 memory locations on the video card, whose heart is a 6845 video processor. For this reason the table to which the vector points and which is part of the ROM-BIOS, consists of 16 consecutive bytes that indicate the contents of individual registers for a certain video mode. The first of these 16 bytes is copied into the first register of the 6845, the second byte into the second register, etc. The table in ROM contains a total of four 16-byte entries: 40x25 color mode, 80x25 color mode, 80x25 monochrome mode and one entry for the various color graphics modes.

Do not call this interrupt. If you do, the system will attempt to read the video table as executable code and will crash.

Input No input

Output No output

Interrupt 1EH	BIOS/DOS
Drive table	

Sets a pointer to a table. The vector of this interrupt in the vector table starting at address 0000:0078 stores the offset and segment address of this table. The table itself contains a collection of parameters used by BIOS in disk drive access. BIOS has a table in ROM, but deviates the interrupt vector of interrupt 30 to its own table which allows faster disk access than the BIOS table.

Do not call this interrupt. If you do call it, the system will attempt to read the drive table as executable code and will crash.

Input No input

Output No output

Interrupt 1FH	BIOS/DOS
Character table	

Sets a pointer to a table. The vector of this interrupt in the vector table, starting at address 0000:007C, stores the offset and segment address of this table. The table itself contains character patterns for the characters possessing ASCII codes 128 to 255. BIOS needs this table to display the graphic mode characters on the screen. These characters are displayed by placing the character patterns, which are stored in this table, on the screen as individual pixels.

Since the character patterns for codes 0 to 127 are already stored in a table in ROM-BIOS, this table contains only the character patterns for codes 128 to 255. The DOS GRAFTABL command loads a table for codes 127 to 255 into RAM and points the interrupt vector of interrupt 31 to this table. A user table can be added to display on the screen, in graphic mode, certain characters that are not part of the normal PC character set. The construction of the table requires that eight consecutive bytes define the appearance of the character. The first eight bytes of the table define the appearance of ASCII code 128, the next eight bytes define ASCII code 129, etc. Each set of eight bytes represents the eight lines which denote a character in graphic mode. The eight bits of each byte indicate the eight columns of pixels for each line.

Do not call this interrupt. If you do call it, the system will attempt to read the character table as executable code and will crash.

Input No input

Output No output

EGA/VGA BIOS Functions

EGA and VGA cards enhance the BIOS Video-Interrupt 10H with numerous functions, which provide access to the extended capabilities of the cards. A description of these functions, which are called like the normal BIOS-functions of the Interrupt 10H, can be found in Appendix B.

Interrupt 10H, Function 00H	EGA/VGA
Screen: Set video mode	

Sets and initializes the video mode.

Input

AH = 00H

AL = EGA/VGA video mode

AL=0: 40x25-character text, 16 colors (EGA/VGA - color monitor)

AL=1: 40x25-character text, 16 colors (EGA/VGA - color monitor)

AL=2: 80x25-character text, 16 colors (EGA/VGA - color monitor)

AL=3: 80x25-character text, 16 colors (EGA/VGA - color monitor)

AL=4: 320x200 graphics, 4 colors (EGA/VGA - color monitor)

AL=5: 320x200 graphics, 4 colors (EGA/VGA - color monitor)

AL=6: 640x200 graphics, 2 colors (EGA/VGA - color monitor)

AL=7: 80x25-character text, mono (EGA/VGA - mono monitor)

AL=13: 320x200 graphics, 16 colors (EGA/VGA - color monitor)

AL=14: 640x200 graphics, 16 colors (EGA/VGA - color monitor)

AL=15: 640x350 graphics, mono (EGA/VGA - mono monitor)

AL=16: 640x350 graphics, 4 colors (64K EGA - high-res monitor)

640x350 graphics, 16 colors (128K EGA/VGA - high-res monitor)

AL=17: 640x480 graphics, 2 colors (VGA only)

AL=18: 640x480 graphics, 16 colors (VGA only)

AL=19: 320x200 graphics, 256 colors (VGA only)

Output

No output

Remarks

Modes 0 and 1, 2 and 3, 4 and 5 differ in the output of the color signal that is suppressed in the first mode. This isn't possible on an EGA/VGA card so the modes are identical. If bit 7 of the AL register is set when this function is called, the contents of the video RAM will not be erased when the new mode is enabled. The task is to program the video controller and define a color palette. The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 01H	EGA/VGA
Screen: Define cursor appearance	

Defines the starting and ending lines of the screen cursor. This function is independent of the .i.display page; being displayed.

Input	AH = 01H
	CH = Starting line of the cursor
	CL = Ending line of the cursor

Output	No output
---------------	-----------

Remarks:

Since the possible values depend on the size of the current video mode's character matrix, the values in the CH and CL registers always refer to an eight-line character matrix. The values should thus be between zero and seven. The EGA/VGA-BIOS adapts these values to the current size of its own character matrix.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 02H	EGA/VGA
Screen: Position cursor	

Moves the cursor into position on the screen.

Input	AH = 02H
	BH = Video page number
	DH = Screen line
	DL = Screen column

Output	No output
---------------	-----------

Remarks:

The cursor moves only if the specified display page is the current page.

The values for the screen line and column are based on the resolution of the current display mode.

Assigning the DH and DL registers values for a non-existent screen position (e.g., column 0, line 255) makes the cursor disappear from the screen.

The number of the display page is based on how many display pages the card has available.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 03H	EGA/VGA
Screen: Read cursor position	

Reads the position of the text cursor on the screen and the starting and ending lines of the screen cursor.

Input	AH = 03H
	BH = Video page number
Output	DH = Screen line in which cursor is located
	DL = Screen column in which cursor is located
	CH = Starting line of screen cursor
	CL = Ending line of screen cursor

Remarks:

The screen line and screen column parameters refer to the text coordinate system, even if a graphic mode is active.

The starting and ending lines of the cursor are returned correctly only in the text modes. They have no meanings in graphic modes.

The contents of registers BX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 04H	EGA/VGA
Screen: Read lightpen position	

Reads the lightpen's position on the screen. This is only possible on EGA cards, since VGA cards do not support lightpens.

Input	AH = 04H
Output	AH = 0: Lightpen position cannot be read
	AH = 1: Lightpen position read
	DH = Screen row (text mode)
	DL = Screen column (text mode)
	CH = Screen row (graphic mode)
	BX = Screen column (graphic mode)

Remarks:

This function call must be repeated until a 1 is returned in the AH register, since only then can the lightpen position be determined.

Graphic mode returns lightpen coordinates less accurately than in text mode. The Y-coordinate (row) is always a multiple of 2. For example, function 04H cannot differentiate between row 8 or row 9. In the 320x200 pixel graphic mode, the X-coordinate (column) is always a multiple of 4, and in the 640x200 pixel graphic mode the X-coordinate is a multiple of 8.

The contents of the CL register and the contents of the segment registers SS, CS and DS are not affected by this function. The content of all other registers may change, especially the SI and DI registers.

Interrupt 10H, Function 05H	EGA/VGA
Screen: Select current display page	

Selects the current display page, and thereby the page which appears on the screen (text mode only).

Input AH = 05H
 AL = Display page number

Output No output

Remarks:

The number of available display pages depends on the amount of video RAM installed on the EGA/VGA card.

When a new page is selected the screen cursor will be moved to the position of the text cursor on this page.

Switching between different pages does not change the contents of these pages.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 06H	EGA/VGA
Screen: Scroll text lines up	

Scrolls part of the current display page up by one or more lines.

Input AH = 06H
 AL = Number of lines to be scrolled up
 AL=0: Clear window
 CH = Screen line of upper left corner of window
 CL = Screen column of upper left corner of window
 DH = Screen line of lower right corner of window
 DL = Screen column of lower right corner of window
 BH = Color (attribute) for blank line(s)

Output No output

Remarks:

Normally the contents of the current display page are scrolled, but in the 320x200 four-color graphic mode this function only affects display page 0.

Clearing the screen window (number of lines = 0) is the same as filling it with spaces (ASCII code 32).

The contents of the lines scrolled out of the window are lost and cannot be recovered.

Use function 0 of this interrupt to clear the screen.

Appendix B: EGA/VGA BIOS Functions

B - 5

The interpretation of the attribute byte in the BL register depends on the current video mode. In text mode it is interpreted as any other attribute byte in video RAM. In 640x200 two-color mode this byte represents the color value for eight successive pixels. In 320x200 four-color mode this byte represents the color value of four successive pixels. In all other graphic modes it represents the color of all the pixels in the cleared screen area.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 07H

EGA/VGA

Screen: Scroll text lines down

Scrolls part of the current display page down one or more lines.

Input

AH = 07H

AL = Number of lines to be scrolled down

AL=0: Clear window

CH = Screen line of upper left corner of window

CL = Screen column of upper left corner of window

DH = Screen line of lower right corner of window

DL = Screen column of lower right corner of window

BH = Color (attribute) for blank line(s)

Output

No output

Remarks:

Normally the contents of the current display page are scrolled, but in 320x200 four-color graphic mode this function only affects display page 0.

Clearing the screen window (number of lines = 0) is the same as filling it with spaces (ASCII code 32).

The contents of the lines scrolled out of the window are lost and cannot be recovered.

To clear the entire screen, use function 0 of this interrupt instead.

The interpretation of the attribute byte in the BL register depends on the current display mode. In the text mode it is interpreted like any other attribute byte in the video RAM. In the 640x200 two-color mode this byte represents the color value for eight successive pixels. In the 320x200 four-color mode it represents the color value of four successive pixels. In all other graphic modes it represents the color of all the pixels in the cleared screen area.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 08H

EGA/VGA

Screen: Read character/color

Reads and returns the ASCII code and color (attribute) of the character at the current cursor position.

Input

AH = 08H

BH = Video page number

Output AL = ASCII code of character

AH = Color (attribute)

Remarks:

This function can also be called in the graphic mode, whereby the bit pattern of the character on the screen will be compared with the bit patterns of the characters. If the character cannot be identified, the AL register will contain the value zero after the call.

In the 320x200 four-color graphic mode this function only affects display page 0.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 09H	EGA/VGA
Screen: Write character/color	

Writes character with the specified color at the current cursor position (in a specified display page).

Input AH = 09H
 BH = Video page number
 CX = Repeat factor
 AL = ASCII code of character
 BL = Attribute

Output No output

Remarks:

If the graphic mode is active and the specified character is to be printed more than once (the value of the CX register is greater than 1), all the characters must fit on the current screen line.

In the 320x200 four-color graphic mode this function correctly works only on display page 0.

Within a graphic mode the attribute in the BL register specifies the foreground color of the character, whereby the background color is zero. If bit seven is set, the character will be XORed with the bitmap at the output position.

The controls codes for bell, carriage return, etc. are not recognized as control codes, and are displayed as normal ASCII characters.

This function can also be used to output characters in the graphic mode, in which case the character patterns are taken from one of the EGA character tables.

This function does not move the cursor to the next screen position.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 0AH	EGA/VGA
Screen: Write character	

Writes a character to the current screen position on the specified display page and the color of the old character at this position will be retained.

Input

AH = 0AH
AL = ASCII code of the character
BH = Video page number
BL = Foreground color of character for graphic modes
CX = Repeat factor

Output

No output

Remarks:

If the graphic mode is active and the specified character is to be printed more than once (the value of the CX register is greater than 1), all the characters must fit on the current screen line.

The controls codes for bell, carriage return, etc. are not recognized as such and are displayed as normal ASCII characters.

This function can also be used to output characters in the graphic mode, in which case the character patterns are taken from one of the EGA character tables.

Within a graphic mode the attribute in the BL register specifies the foreground color of the character, whereby the background color is zero. If bit seven is set, the character will be XORed with the bitmap at the output position.

This function does not move the cursor to the next screen position.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 0BH, Subfunction 0**EGA/VGA**

Screen: Select border/background color

Selects the border and background color for the graphic or text mode.

Input

AH = 0BH
BH = 0
BL = Border/background color

Output

No output

Remarks:

This function should be called only when the EGA/VGA card is in the 320x200 or 640x200 graphic mode. Use function 10H for all other modes.

Bits zero to three of the BL register set the background and border color. Setting bit four will enable high-intensity colors.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 0BH, Subfunction 1**EGA/VGA**

Screen: Select color palette

Selects one of the two color palettes for the 320x200 graphic mode.

Input AH = 0BH
 BH = 1
 BL = Color palette number

Output No output

Remarks:

This function should be called only when the EGA/VGA card is in the 320x200 or 640x200 graphic mode. Use function 10H for all other modes.

The EGA/VGA-BIOS emulates the two CGA color palettes with the numbers 0 and 1. They contain the following colors:

Palette 0: green, red, yellow

Palette 1: cyan, magenta, white

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 0CH	EGA/VGA
Screen: Write pixel	

Sets the color value of a screen pixel in the graphic mode.

Input AH = 0CH
 BH = Video page
 DX = Screen line
 CX = Screen column
 AL = Color value

Output No output

Remarks:

The color value depends on the colors available in the current display mode.

If bit seven of the AL register is set, the color value will be XORed with the previous color value of the pixel.

The display page is ignored in the 320x200 four-color graphic mode.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 0DH	EGA/VGA
Screen: Read pixel	

The color value of a pixel in the graphic mode is returned.

Input AH = 0DH
 BH = Video page

DX = Screen line

CX = Screen column

Output AL = Color value

Remarks:

The color value depends on the colors available in the current display mode.

The display page is ignored in the 320x200 four-color graphic mode.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 0EH

EGA/VGA

Screen: Write character

Writes a character to the current cursor position on the current display page. The color of the old character at this position will be retained.

Input AH = 0EH

AL = ASCII character code

BL = Foreground color of character

Output No output

Remarks:

This function does not treat the various control codes like bell and carriage as normal characters, and implements them as the control characters they represent.

After displaying a character with this function, the cursor position is incremented so the next character will be printed at the following screen position. If the last screen position has been reached, the screen will be scrolled up one line and the output will continue in the first column of the last screen line.

If bit seven of the BL register is set, the color value will be XORed with the previous color value of the pixels. The background color is zero.

Characters can be displayed in the graphic mode with this function. The character patterns are taken from one of the EGA character tables.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 0FH

EGA/VGA

Screen: Returns current display mode

Reads the number of the current display mode, the number of characters per line, and the number of the current display page.

Input AH = 0FH

Output	AL = Video mode:
	AL=0: 40x25-character text, 16 colors (EGA/VGA - color monitor)
	AL=1: 40x25-character text, 16 colors (EGA/VGA - color monitor)
	AL=2: 80x25-character text, 16 colors (EGA/VGA - color monitor)
	AL=3: 80x25-character text, 16 colors (EGA/VGA - color monitor)
	AL=4: 320x200 graphics, 4 colors (EGA/VGA - color monitor)
	AL=5: 320x200 graphics, 4 colors (EGA/VGA - color monitor)
	AL=6: 640x200 graphics, 2 colors (EGA/VGA - color monitor)
	AL=7: 80x25-character text, mono (EGA/VGA - mono monitor)
	AL=13: 320x200 graphics, 16 colors (EGA/VGA - color monitor)
	AL=14: 640x200 graphics, 16 colors (EGA/VGA - color monitor)
	AL=15: 640x350 graphics, mono (EGA/VGA - mono monitor)
	AL=16: 640x350 graphics, 4 colors (64K EGA - high-res monitor)
	640x350 graphics, 16 colors (128K EGA/VGA - high-res monitor)
	AL=17: 640x480 graphics, 2 colors (VGA only)
	AL=18: 640x480 graphics, 16 colors (VGA only)
	AL=19: 320x200 graphics, 256 colors (VGA only)
	AH = Number of characters per line
	BH = Number of current display page

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 00H	EGA/VGA
Screen: Set palette registers	

Sets the contents of a palette register in the attribute controller of the EGA/VGA card.

Input	AH = 10H
	AL = 00H
	BL = Color value
	BH = Register to be addressed
Output	No output

Remarks:

Since the register number is not checked by the BIOS, you can also program the other registers in the attribute controller. These include the mode control register, overscan register and others.

The contents of registers BX, CX, DX, SI, DI, BP, and the segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 01H	EGA/VGA
Screen: Set screen border color	

Copies resulting value into the .i.overscan register; of the .i.EGA attribute controller;.

Input	AH = 10H
	AL = 01H
	BH = Border color

Output	No output
---------------	-----------

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 02H	EGA/VGA
Screen: Set all palette registers	

Configures all 16 palette registers and the .i.overscan register;.

Input	AH = 10H
	AL = 02H
	ES = Segment address of color table
	DX = Offset address of color table

Output	No output
---------------	-----------

Remarks:

The ES:BX register pair points to a 17-byte table. The first 16 bytes will be transferred to the 16 palette registers of the attribute controller and the 17th byte will be copied into the overscan register.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 03H	EGA/VGA
Screen: Set blinking attribute	

Determines whether bit 7 in the attribute byte of a character in the text mode will enable character blinking, or display characters on a high-intensity background.

Input AH = 10H
 AL = 00H
 BL = Blinking attribute
 BL=0: high-intensity background
 BL=1: blinking

Output No output

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 07H	VGA
Screen: Read out palette register	

Reads the contents of one of the attribute controller's palette registers.

Input AH = 10H
 AL = 07H
 BH = Number of palette register

Output BL = Contents of addressed palette register

Remarks:

Since the BIOS doesn't verify the number of the palette register read, this function can read all the registers of the attribute controller.

The contents of registers BL, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 08H	VGA
Screen: Read contents of overscan register	

Returns the contents of the overscan register containing the screen's border color.

Input AH = 10H
 AL = 08H

Output BH = Contents of the overscan register

Remarks:

The contents of registers BL, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 09H**VGA**

Screen: Read contents of all palette registers and overscan register

Copies the contents of the 16 palette registers and overscan register into a buffer.

Input

AH = 10H

AL = 09H

ES = Segment address of the buffer

DX = Offset address of the buffer

Output

No output

Remarks:

The buffer must be a minimum of 17 bytes long to allow room for all the palette registers (bytes 0-15) plus the overscan register (byte 16).

The contents of registers BL, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 10H**VGA**

Screen: Define a DAC color register

Allows the definition of one of the 256 available DAC color registers.

Input

AH = 10H

BX = Number of the DAC color register (0-255)

CH = Green value

CL = Blue value

DH = Red value

Output

No output

Remarks:

Only bits 0 to 5 in the CH, CL and DH registers are of importance to this function. All other bits are ignored.

The contents of registers BL, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 12H**VGA**

Screen: Load multiple DAC color registers

Allows the definition of multiple DAC color registers.

Input

AH = 10H

AL = 12H

BX = Number of the first DAC color register (0-255)

CX = Number of registers to be loaded

ES = Segment address of the buffer

DX = Offset address of the buffer

Output No output

Remarks:

The assigned buffer must be able to hold a group of three consecutive bytes for each DAC color register. The first byte contains the red value; the second byte contains the green value; and the third byte contains the blue value. These first three bytes correspond to the first DAC color register being accessed, the next three for the bytes to the next DAC color register.

Only bits 0 to 5 in the CH, CL and DH registers are of importance to this function. All other bits are ignored.

If the sum of BX and CX is greater than 255, the first DAC color register is reloaded after the last register is loaded.

The contents of registers BL, CX, DX, SI, DI, BP and all segment registers are unchanged by this function.

Interrupt 10H, Function 10H, Subfunction 13H

VGA

Screen: Select color register or select a DAC register group

Manipulates bit 7 of the mode control registers.

Input AH = 10H

AL = 13H

BL = 00H or 01H (see below)

BH = see below

Output No output

Remarks:

This subfunction performs as two different subfunctions, depending on the value contained in the BL register. Subfunction 00H allows color selection, while subfunction 01H allows the selection of the active DAC register group.

Subfunction 00H copies bit 0 in the BH register into bit 7 of the mode control register, thus providing a method of color selection. If bit 0 in the BH register contains a value of 0, then the 256 DAC color registers are divided into four groups of 64 registers. Color selection involves bits 0-5 in the corresponding palette register, as well as bits 2-3 of the color select register. These eight bits act as the index for the DAC color register. If bit 0 in the BH register contains a 1, the DAC color registers are divided into 16 groups of 16 registers. Then color selection involves the lowest 4 bits of the palette register and the lowest 4 bits of the color select register, acting as the 8-bit index to the DAC color table.

Subfunction 01H loads the color select register, whose contents are specified by the active group of DAC color registers. The contents of the BH register are copied to the color select register.

The contents of registers BL, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 15H	VGA
Screen: Read a DAC color register	

Returns the contents of one of the 256 DAC color registers.

Input	AH = 10H
	AL = 15H
	BX = Number of the DAC color registers

Output	CH = Green value
	CL = Blue value
	DH = Red value

Remarks:

Only bits 0 to 5 in the CH, CL and DH registers are of importance to this function. All other bits are ignored.

The contents of registers BX, DL, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 17H	VGA
Screen: Load contents of multiple DAC color registers	

Loads several DAC color registers at a time.

Input	AH = 10H
	AL = 17H
	BX = Number of the first DAC color register to be loaded (0-255)
	CX = Number of registers to be loaded
	ES = Segment address of buffer
	DX = Offset address of buffer

Output	No output
---------------	-----------

Remarks:

The contents of each DAC color register are represented within a buffer by three consecutive bytes. The red value is loaded into the first of these registers; the green value is loaded into the second of these registers; and the blue value is loaded into the third register. The first group of three bytes corresponds to the first DAC color register addressed, the second group to the next DAC color register, etc.

Only bits 0 to 5 in the CH, CL and DH registers are of importance to this function. All other bits are ignored.

If the sum of BX and CX is greater than 255, the first DAC color register is reloaded after the last register is loaded (wrap-around occurs).

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 18H	VGA
Screen: Load DAC mask register	

Loads the specified value into the DAC mask register.

Input	AH = 10H
	AL = 18H
	BL = Value of DAC mask register

Output	No output
---------------	-----------

Remarks:

The contents of the DAC mask register play an important role in color selection. An AND instruction adds it to the index access to the DAC color table.

The contents of registers BH, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 19H	VGA
Screen: read out contents of the DAC mask register	

Reads the current contents of the DAC mask register.

Input	AH = 10H
	AL = 19H
Output	BL = Contents of the DAC mask register

Remarks:

The contents of the DAC mask register play an important role in color selection. An AND instruction adds it to the index access to the DAC color table.

The contents of registers BH, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 1AH	VGA
Screen: Returns method of color selection and color select register	

Returns the method of color selection, contained in the contents of bit 7 of the mode control register. It also returns the contents of the color select register chosen by the active group of DAC color registers.

Input	AH = 10H
	AL = 1AH
Output	BL = Bit 7 of mode control register
	BH = Contents of color select registers

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 10H, Subfunction 1BH	VGA
Screen: Convert DAC color register into gray scales	

Converts a specified range within a DAC color table into gray scales.

Input	AH = 10H
	AL = 1BH
	BX = Number of first DAC color register to be converted
	CX = Total number of DAC color registers to be converted

Output	No output
---------------	-----------

Remarks:

Conversion into grays results from changes to the red, green and blue values, as well as the intensity of these values. The default factor for red is 0.3, the default factor for green is 0.59, and the default for blue 0.11.

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 00H	EGA/VGA
Screen: Load user-defined character set	

Loads a user-defined character set from RAM into one of the two EGA character tables.

Input	AH = 11H
	AL = 00H
	BH = Lines per character (also bytes per character)
	BL = Character table (0 or 1)
	CX = Number of characters in table
	DX = ASCII code of first character in table
	ES = Segment address of character table in RAM
	BP = Offset address of character table in RAM

Output	No output
---------------	-----------

Remarks:

A maximum of 512 characters can be loaded per character table.

The loaded character set is not activated, nor are the .i.CRTC; registers programmed to the size of the characters. The changes will not be visible on the screen unless the character definitions are loaded into the active character table.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 01H	EGA/VGA
Screen: Load 8x14 character set	

Loads the entire 8x14-pixel character set from EGA/VGA ROM-BIOS into one of the two character set tables.

Input

AH = 11H

AL = 01H

BL = Character table (0 or 1)

Output No output

Remarks:

The loaded character set is not activated, nor are the CRTC registers programmed to the size of the characters. The changes will not be visible on the screen unless the character definitions are loaded into the active character table.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 02H	EGA/VGA
Screen: Load 8x8 character set	

Loads the entire 8x8-pixel character set from EGA/VGA ROM-BIOS into one of the two character set tables.

Input

AH = 11H

AL = 02H

BL = Character table (0 or 1)

Output No output

Remarks:

The loaded character set is not activated, nor are the CRTC registers programmed to the size of the characters. The changes will not be visible on the screen unless the character definitions are loaded into the active character table. The EGA card displays 43 lines on the screen, while the VGA card displays 50 lines.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 03H	EGA/VGA
Screen: Activate character set	

Activates one (or two) of the four 256-character character sets.

Input

AH = 11H

AL = 03H

BL = Number of the character set to activate

Output No output

Remarks:

Bits zero and one of the BL register specify the number of the character set to be accessed when bit three of the attribute byte of the character is zero.

Bits two and three of the BL register specify the number of the character set to be accessed when bit three of the attribute byte of the character is one.

If the contents of bits zero and one are identical to the contents of bits two and three of the BL register, then bit three of the character attribute byte has no effect on the character displayed. Only 256 different characters can then be displayed on the screen.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 04H

VGA

Screen: Load 8x16 character set

Loads the entire 8x16-pixel character set from the VGA-BIOS into one of the two character set tables.

Input

AH = 11H

AL = 04H

BL = Corresponding character set table (0 or 1)

Output No output

Remarks:

The loaded character set is not activated, nor are the CRTC registers programmed to the size of the characters. The changes will not be visible on the screen unless the character definitions are loaded into the active character table. The VGA card displays 25 text lines on the screen.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 10H

EGA/VGA

Screen: Load and activate user-defined character set

Loads a user-defined character set from RAM into one of the two EGA character tables and activates it by programming the CRTC registers.

Input

AH = 11H

AL = 10H

BH = Lines per character (also bytes per character)

BL = Character table (0 or 1)

CX = Number of characters in table

DX = ASCII code of first character in table

ES = Segment address of character table in RAM

BP = Offset address of character table in RAM

Output No output

Remarks:

A maximum of 512 characters can be loaded per character table.

The number of text lines displayed on the screen results from the height of the individual characters. It is calculated by dividing the number of screen lines (350) by the character height.

The starting and ending lines of the screen cursor are automatically adapted to the height of the new character matrix.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 11H	EGA/VGA
Screen: Load and activate 8x14 character set	

Loads the entire 8x14-pixel character set from EGA/VGA ROM-BIOS into one of the two character set tables, and activates it by programming the CRTC registers.

Input AH = 10H

AL = 11H

BL = Character table (0 or 1)

Output No output

Remarks:

The function sets the EGA screen to display 25 lines of text, or sets the VGA screen to display 28 lines of text.

The starting and ending lines of the screen cursor are automatically adapted to the height of the new character matrix.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 12H	EGA/VGA
Screen: Load and activate 8x8 character set	

Loads the entire 8x8-pixel character set from the ROM-BIOS of the EGA/VGA card into one of the two character set tables, and activates it by programming the CRTC registers.

Input AH = 10H

AL = 12H

BL = Character table (0 or 1)

Output No output

Remarks:

The function sets the screen to display 43 lines of text (EGA) or 50 lines of text (VGA).

The starting and ending lines of the screen cursor are automatically adapted to the height of the new character matrix.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 14H	VGA
Screen: Load 8x16 character set	

Loads a complete 8x16 character set from the VGA card BIOS into one of the two character set tables, and activates it through CRTC register programming.

Input

AH = 10H

AL = 14H

BL = Character table (0 or 1)

Output No output

Remarks:

When this function is called, the VGA card displays 25 lines of text on the screen.

The starting and ending lines of the screen cursor automatically change to match the height of the new character matrix.

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 11H, Subfunction 30H	EGA/VGA
Screen: Get information about the character generator	

Returns various information about the current status of the character generator.

Input

AH = 11H

AL = 03H

BH = Type of information desired

BH=0: contents of interrupt vector 1FH

BH=1: contents of interrupt vector 43H

BH=2: address of the ROM 8x14 character table

BH=3: address of the ROM 8x8 character table

BH=4: address of the second half of the 8x8 character table

BH=5: address of the alternative ROM 9x14 character table

BH=6: Address of the alternative ROM 8x16 character table

BH=7: Address of the alternative ROM 9x16 character table

Output	CX = Height of current character matrix
	DL = Number of columns per line - 1
	ES = Segment address of the pointer
	BP = Offset address of the pointer

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers CS, DS and SS are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 10H	EGA/VGA
Screen: Determine EGA/VGA configuration	

Reads the configuration of the EGA/VGA card.

Input	AH = 12H
	BL = 10H
Output	BH = Monitor connected
	BH=0: color or high-resolution monitor
	BH=1: monochrome monitor
	BL = EGA/VGA RAM capacity
	BL=0: 64K
	BL=1: 128K
	BL=2: 192K
	BL=3: 256K

Remarks:

The contents of registers DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 20H	EGA/VGA
Screen: Activate alternate hardcopy routine	

Installs an alternative hardcopy routine which prints as many lines as are displayed on the screen. The hardcopy routine of the normal ROM-BIOS always prints 25 lines and is not suited for creating a hardcopy of the EGA/VGA modes, which display more than 25 lines on the screen.

Input	AH = 12H
	BL = 20H
Output	No output

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 30H	EGA/VGA
Screen: Specify number of scan lines	

Selects the number of scan lines on the screen.

Input	AH = 12H
	BL = 30H
	AL = Scan line status
	AL=0: 200 scan lines (EGA and VGA)
	AL=1: 350 scan lines (EGA and VGA)
	AL=2: 400 scan lines (VGA only)

Output	No output
---------------	-----------

Remarks:

The selected number of scan lines can only be displayed when the appropriate video card and monitor are in use. For example, a CGA monitor can only display 200 scan lines, even if the video card can operate in a higher resolution.

The contents of registers BX, CX, DX, SI, DI and BP and all segment registers are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 31H	VGA
Screen: Toggle palette registers loading	

Toggles the automatic loading of palette registers in VGA-BIOS. The system either loads alternate display modes when function 00H is invoked, or loads default values.

Input	AH = 12H
	BL = 31H
	AL = Automatic palette register loading
	AL=0: Yes
	AL=1: No

Output	No output
---------------	-----------

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 32H	EGA/VGA
Screen: Enable/disable CPU access to video RAM	

Enables or disables direct CPU access to video RAM and its different I/O ports.

Input

AH = 12H

BL = 32H

AL = Access status

AL=0: Access enabled

AL=1: Access denied

Output No output

Remarks:

The EGA BIOS doesn't recognize this function, but you can still suppress video card access directly using bit 1 of the output register (port address 3C2H).

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 33H	VGA
Screen: Enable/disable automatic gray scaling in DAC color registers	

Toggles automatic gray scaling in VGA-BIOS. This is different from function 10H, subfunction 1BH, which enables selective gray scaling in DAC color registers.

Input

AH = 12H

BL = 33H

AL = DAC color register gray scaling

AL=0: On

AL=1: Off

Output No output

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 34H	VGA
Screen: Enable/disable text cursor emulation	

Toggles text cursor emulation mode. Calling function 01H (for defining the starting and ending lines of the cursor) doesn't compensate for character matrices in different resolutions. This function controls that change when in VGA mode.

Input

AH = 12H

BL = 34H

AL = Cursor emulation mode

AL=0: On

AL=1: Off

Output No output

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 12H, Subfunction 36H

VGA

Screen: Suppress screen refresh

Temporarily suppresses screen refresh. Disabling refresh relieves video RAM of many system level tasks, especially those involving complex screen graphics.

Input AH = 12H

BL = 36H

AL = Screen refresh

AL=0: On

AL=1: Off

Output No output

Remarks:

The contents of registers BX, CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 13H

EGA/VGA

Screen: Display a string

Displays a string at a specified position on the screen, in a specific display page. The characters are taken from a buffer whose address is passed to the function.

Input AH = 13H

AL = Output mode (0-3)

AL=0: Attribute in BL, reserve cursor position

AL=1: Attribute in BL, update cursor position

AL=2: Attributes in buffer, reserve cursor position

AL=3: Attributes in buffer, update cursor position

BL = Attribute byte of characters (modes 0 and 1 only)

CX = Number of characters to be printed

DH = Screen line
 DL = Screen column
 BH = Video page
 ES = Segment address of the buffer
 BP = Offset address of the buffer

Output No output

Remarks:

In modes 1 and 3 the cursor position is placed after the last character of the string so that BIOS output will continue at the character after the string. This does not happen in modes 0 and 2.

In modes 0 and 1 the buffer contains only the ASCII codes of the characters to be printed. The color of all the characters in the string is specified by the BL register. In modes 2 and 3, each character in the buffer is followed by the corresponding attribute byte, so that each character has its own attribute. The BL register does not have to be loaded in these modes. Although the string must be twice as long as the number of characters to be printed in these modes, the CX register contains just the number of ASCII characters to be printed, not the string buffer's length.

Control codes such as bell and carriage return are interpreted as control codes and not as normal ASCII codes. An error occurs when carriage return and linefeed are printed on a display page other than zero, however. These characters may be printed on display page 0, regardless of the display page specified in BH.

When the last screen position is reached the screen will move up one line and the output will continue with the first column of the last screen line.

When printing in the graphic mode the contents of the BL register determine the foreground color of the character (the background is zero). If bit seven of the BL register is set, the color value will be XORed with the old color value.

This function can also be used to print characters in the graphic mode, in which case the character patterns will be taken from one of the EGA/VGA character tables.

The contents of registers BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 10H, Function 14H

EGA/VGA

Undocumented

Interrupt 10H, Function 15H

EGA/VGA

Undocumented

Interrupt 10H, Function 16H

EGA/VGA

Undocumented

Interrupt 10H, Function 17H	EGA/VGA
Undocumented	

Interrupt 10H, Function 18H	EGA/VGA
Undocumented	

Interrupt 10H, Function 19H	EGA/VGA
Undocumented	

Interrupt 10H, Function 1AH	VGA
Screen: Determine video card type	

Determines the existence of the active video card.

Input	AH = 1AH
	AL = 0
Output	AL = 1AH
	BL = Device code for active video card
	BH = Device code for inactive video card

Remarks:

If the value 1AH is not loaded into the AL register, then the video card in operation is not a VGA card (the 1AH indicates a VGA-BIOS). The function can return the following device codes:

- FFH = Unknown video card
- 00H = No video card
- 01H = MDA with monochrome display
- 02H = CGA with CGA monitor
- 04H = EGA with EGA or multisync monitor
- 05H = EGA - monochrome display
- 07H = VGA - analog monochrome display
- 08H = VGA - analog color display (VGA, multisync)

The contents of registers CX, DX, SI, DI, BP and all segment registers are not affected by this function.

Interrupt 10H, Function 1BH	VGA
Get VGA-BIOS and video mode status	

Gets information about the current VGA-BIOS and video card status.

Input

AH = 1BH

BX = 0

ES:DI = Pointer to a buffer

Output

AL = 1BH

Remarks:

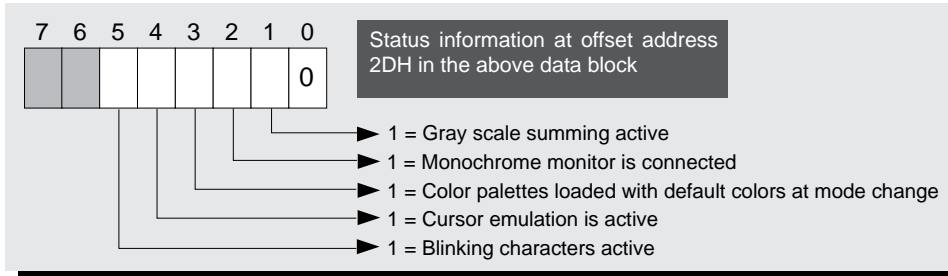
If the value 1BH is not loaded into the AL register, then the video card in operation is not a VGA card (the 1BH indicates a VGA-BIOS).

The buffer passed during the call of this function must have a minimum capacity of 64 bytes, because the VGA-BIOS stores a table there which contains information describing the current video mode.

The contents of registers BX, CX, DX, SI, DI and BP and all segment registers are not affected by this function.

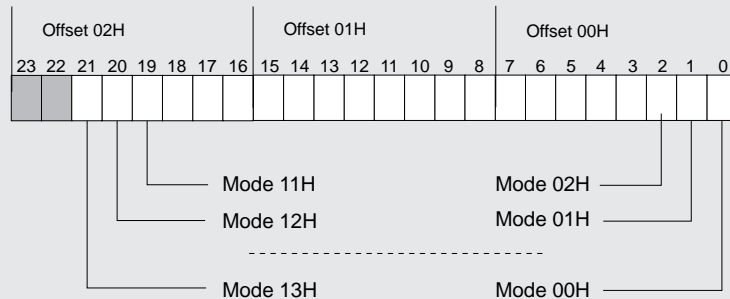
Data block with mode dependent VGA-BIOS status information		
Off	Content	Type
00H	Address of table containing static information	1 ptr
04H	Code number of current video mode	1 byte
05H	Number of displayed screen or pixel columns	1 word
07H	Length of display page in video RAM	1 word
09H	Starting address of current display page in video RAM	1 word
0BH	Cursor positions in maximum of eight display pages in column/row order	16 bytes
1BH	Ending row of cursor (pixel row)	1 byte
1CH	Starting row of cursor (pixel row)	1 byte
1DH	Number of current display page	1 byte
1EH	Port address of the CRT controller address register	1 word
20H	Current contents of CRTC control registers at port address 3B8H (MDA emulation) or 3D8H (VGA)	1 byte
21H	Current color selection register contents at port address 3D9H	1 byte
22H	Number of screen rows displayed	1 byte
23H	Height of characters in pixel rows	1 word
25H	Code number of active video adapter (see function 1AH, subfunction 00H)	1 byte
26H	Code number of inactive video adapter (see function 1AH, subfunction 00H)	1 byte
27H	Number of displayable colors (0 = monochrome)	1 word

Data block with mode dependent VGA-BIOS status information (continued)		
Off	Content	Type
29H	Number of screen pages	1 byte
2AH	Number of displayed pixel rows 0 = 200 pixel rows 1 = 350 pixel rows 2 = 400 pixel rows 3 = 480 pixel rows	1 byte
2BH	Number of character table used with characters whose	1 byte
2CH	Number of character table used with characters whose third bit of the attribute byte is 1	1 byte
2DH	Miscellaneous information #1 (see below)	1 byte
2EH	Reserved	3 bytes
31H	Size of available video RAM	1 byte
	0 = 64K 1 = 128K 2 = 192K 3 = 256K	
32H	Reserved	14 bytes
64 bytes		

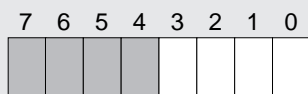


Data block with mode independent VGA-BIOS status information		
Off.	Content	Type
00H	Address of table containing static information	1 ptr
00H	Supported video modes (see below)	3 bytes
03H	reserved	4 bytes
07H	Number of pixel rows in text mode (see below)	1 byte
08H	Maximum number of character sets that can be displayed	1 byte
09H	Number of character sets loadable into video RAM	1 byte
0AH	VGA-BIOS capability information (see below)	1 byte
0BH	More VGA-BIOS capability information (see below)	1 byte
0CH	Reserved	2 bytes
0EH	Reserved	2 bytes
16 bytes		

Video mode support information found at offset address 00H as listed in the above data block

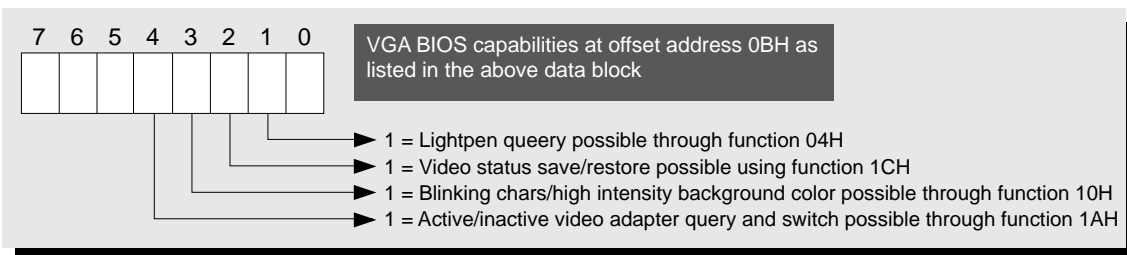
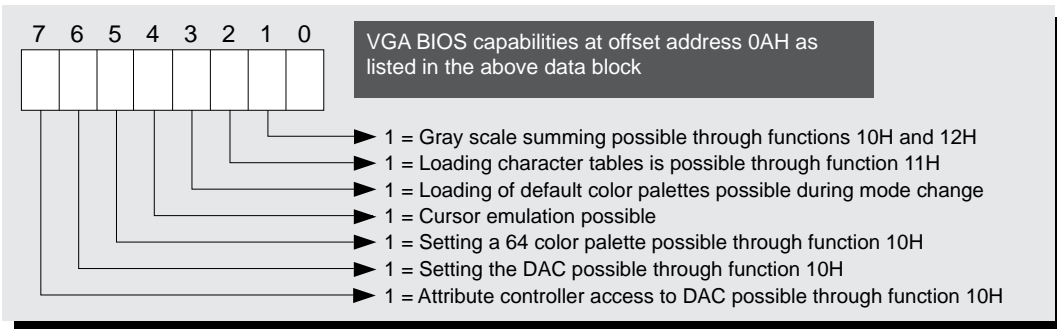


Bit contains 00000001 if corresponding video mode is supported



Pixel row information found at offset address 07H as listed in the above data block

- 1 = 200 pixel rows possible
- 1 = 350 pixel rows possible
- 1 = 400 pixel rows possible
- 1 = 480 pixel rows possible

**Interrupt 10H, Function 1CH, Subfunction 00H****VGA**

Return save/restore status

Returns information about the save/restore buffer.

Input

AH = 1CH

AL = 0

CX = Components to be saved

Bit 0=Video hardware

Bit 1=Data areas of VGA-BIOS

Output

AL = 1CH

BX = Buffer size in units of 64 bytes

Remarks:

If the value 1CH is not returned in the AL register, no VGA-BIOS exists, indicating that no VGA card is installed.

During the function call, the three lowest bits in the CX register mark the video hardware components and the VGA-BIOS which should be stored. The bit is set to 1 if the corresponding component should be stored.

Video RAM is not considered part of video hardware, and must be stored separately by programs.

The contents of the CX, DX, SI, DI and BP registers and all segment registers are not affected by this function.

Interrupt 10H, Function 1CH, Subfunction 01H	VGA
Store video status	

Stores video hardware/VGA-BIOS status received after calling subfunction 00H.

Input AH = 1CH
 AL = 1
 CX = Components to be stored
 Bit 0=Video hardware
 Bit 1=Data areas of VGA-BIOS
 ES:BX = Pointer to buffer in which information should be stored

Output AL = 1CH

Remarks:

If the value 1CH is not returned in the AL register, no VGA-BIOS exists, indicating that no VGA card is installed.

Subfunction 00H must be used to determine the buffer size required before calling this function. Notice the contents of the CX register remains unchanged between the two calls, During the function call, the three lowest bits in the CX register mark the video hardware components and the VGA-BIOS which should be stored. The bit is set to 1 if the corresponding component should be stored. Video RAM is not considered part of video hardware, and must be stored separately by programs.

The contents of the BX, CX, DX, SI, DI and BP registers and all segment registers are not affected by this function.

Interrupt 10H, Function 1CH, Subfunction 02H	VGA
Restore video status	

Restores video hardware/VGA-BIOS status stored by a preceding call to subfunction 01H.

Input AH = 1CH
 AL = 2
 CX = Components to be restored
 Bit 0=Video hardware
 Bit 1=Data areas of VGA-BIOS
 ES:BX = Pointer to buffer in which information was previously stored

Output AL = 1CH

Remarks:

If the value 1CH is not returned in the AL register, no VGA-BIOS exists, indicating that no VGA card is installed. This function call makes sense only following a call to subfunction 01H, where no more components can be restored than were stored previously. Before the function call note the content of the CX register.

The contents of the BX, CX, DX, SI, DI and BP registers and all segment registers are not affected by this function.

VESA Standard Functions

The VESA interface makes standardized functions available for access to Super VGA cards. This interface allows a program to connect with the various Super VGA cards, even though these cards are isolated from programs. The six different VESA standard functions are subfunctions of function 4FH, which links the VESA driver (or VESA compatible VGA BIOS) to BIOS interrupt 10H.

VESA BIOS Graphic Modes			
Code	Resolution	Colors	Memory
100H	640x 400	256	256K
101H	640x 480	256	512K
102H	800x 600	16	256K
103H	800x 600	256	512K
104H	1024x 768	16	512K
105H	1024x 768	256	1 Meg
106H	1280x1024	16	1 Meg
107H	1280x1024	256	1.25 Meg
6AH	800x 600	16	256 K

Interrupt 10H, Function 4FH, Subfunction 00H	VESA
Get Super VGA card information	

Reads the capabilities of the active Super VGA card, and determines which VESA functions are supported by that card

Input	AH = 4FH
	AL = 00H
	ES:DI = FAR pointer to the info buffer
Output	AL = 4FH: VESA functions are supported
	AH = 00H: VESA functions are supported

Remarks

The info buffer cited in the ES:DI registers must have 256 bytes of memory available. If the function executes correctly, it contains the following information after the call:

Structure of the Info Buffer		
Offset	Content	Type
00H	VESA signature ("VESA")	4 byte
04H	VESA version (higher level version number)	1 byte
05H	VESA version (lower level version number)	1 byte
06H	FAR pointer to ASCII string containing the name of card's manufacturer	1 dword
0AH	Flag, indicating capabilities of card (not used: Flag = 0000H)	1 dword
0EH	FAR pointer to list of code numbers indicating supported video modes	1 dword

The list with the code numbers of the supported video modes, which are passed in the last field of the buffer, consists of various words, which indicate the code of a video mode according to the table above. This list is terminated by a word containing the value 0FFFFH. The length of the list varies from card to card.

Interrupt 10H, Function 4FH, Subfunction 01H	VESA
Query data for a VESA mode	

This function returns information about a VESA mode, but does not switch it on.

Input	AH = 4FH
	AL = 01H
	CX = Code number of the desired VESA mode
	ES:DI = FAR pointer to info buffer
Output	AL = 4FH and
	AH = 00H : function was executed in an orderly manner

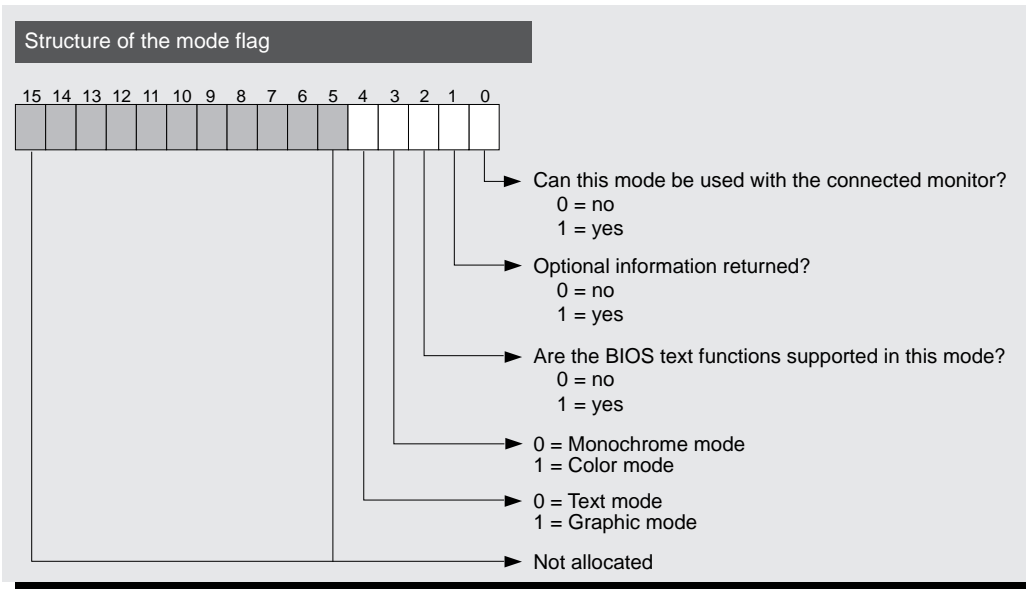
Remarks

This function should only be called, after the subfunction 00H was called successfully and therefore the existence of a VESA driver has been proven. Also only modes can be queried which appear in the mode list of function 00H.

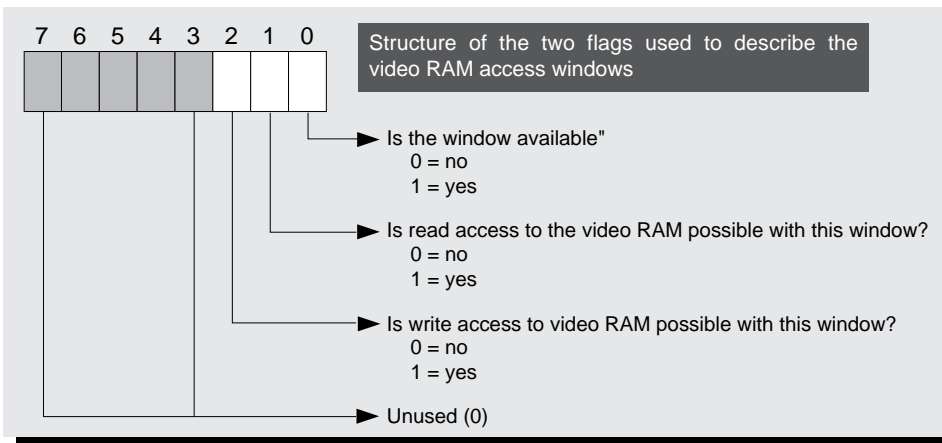
The info-buffer, whose address is passed to the function, must offer 29 bytes of storage space. If the function returns successfully to the caller, it contains the following information:

The Info-Buffer		
Offs	Content	Type
00H	Mode-Flag, see below	1 WORD
02H	Flags for the first access window, see below	1 BYTE
03H	Flags for the second access window, see below	1 BYTE
04H	Granularity in K, with which the two access windows can be moved	1 WORD
06H	Size of the two access windows in KB	1 WORD
08H	Segment address of the first access window	1 WORD
0AH	Segment address of the second access window	1 WORD
0CH	FAR-pointer to the routine for setting of the visible area in the two access windows	1 DWORD
10H	Number of bytes, occupied by the individual dot line in Video-RAM	1 WORD
Optional information, see mode-Flag		
12H	X-resolution in dots/characters	1 WORD
14H	Y-resolution in dots/characters	1 WORD
16H	Width of the der character matrix in dots	1 BYTE
17H	Height of the character matrix in dots	1 BYTE
18H	Number of Bit-Planes	1 BYTE
19H	Number of bits per screen dot	1 BYTE
1AH	Number of storage blocks	1 BYTE
1BH	Storage model	1 BYTE
1CH	Size of the storage blocks in KB	1 BYTE

The mode flag (offset 00H) tells if the optional fields in the info buffer of the VESA function were filled out and returns important information about the desired mode.



The two access windows (offset 02H and 03H) are described through the following bit fields:



The storage model (offset 1BH) reflects the structure of the Video RAM in the desired video mode. These codes are known.

Valid Codes for the description of the storage model			
Num.	Task	Num.	Task
00H	Text mode	04H	Packed format with two dots with 4 bits per byte
01H	CGA-Format, i.e. 2 or 4 storage blocks	05H	Normal EGA-/VGA format for 256 color graphic mode
02H	Hercules-Format with 4 storage blocks	06H-0FH	Reserved
03H	Normal EGA-/VGA format for 16 color graphic mode	10H-FFH	Manufacturer specific code, not used until now

Interrupt 10H, Function 4FH, Subfunction 02H	VESA
Switch on VESA mode	

A VESA mode can be switched on by using this function.

Input AH = 4FH
 AL = 02H
 BX = Code number of the desired mode

Output AL = 4FH and
 AH = 00H: function was executed in an orderly manner

Remarks

This function should only be called, after the subfunction 00H was called successfully and therefore the existence of a VESA driver has been proven. Also only modes can be queried which appear in the mode list of function 00H.

Bit 15 in the code number of the video mode in the BX register, can be set when the Video RAM should not be erased during the initialization of the video mode.

Interrupt 10H, Function 4FH, Subfunction 03H	VESA
Query current mode	

This function returns the code number of the current video mode and takes into consideration also the non VESA modes.

Input AH = 4FH
 AL = 03H

Output AL = 4FH and
 AH = 00H : was executed in an orderly manner in this case
 BX = Code-number of the current mode

Remarks

This function should only be called, after the subfunction 00H was called successfully and therefore the existence of a VESA driver has been proven.

Interrupt 10H, Function 4FH, Subfunction 04H/00H	VESA
Determine the size of the safety buffer	

For the call of the subfunction 04H/01H which follows, this function is used to determine the size of the required safety buffer.

Input AH = 4FH
 AL = 04H
 DL = 00H
 CX = Components of the video-status to be stored

Output AL = 4FH and

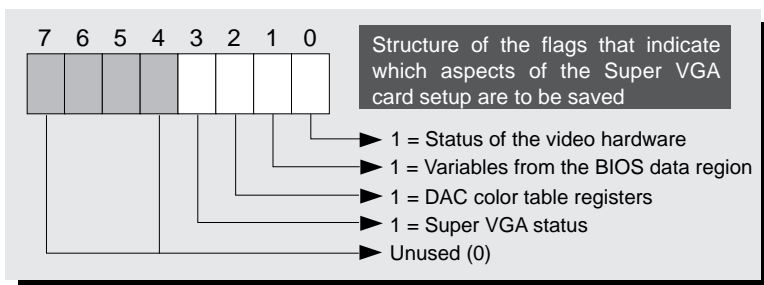
AH = 00H : Function was executed in an orderly manner in this case

BX = Number of consecutive 64 byte blocks which are required as safety buffer

Remarks

This function must be called before the subfunction 04H/01H to determine the size of the safety buffer which is made available to the subfunction 04H/01H.

During the function call, the various bits in the CX register indicate, which components of the video-status should be stored. Only the low byte of the CX register is used, in which the following bits are of significance:

**Interrupt 10H, Function 4FH, Subfunction 04H/01H****VESA**

Store Video-Status of the Super VGA card

After a preceding call of the subfunction 04H/00H, the desired information about the various components of the video status are stored in a buffer of the caller by this function.

Input AH = 4FH

AL = 04H

DL = 01H

CX = Components of the video status to be stored

ES:BX = FAR pointer to safety buffer

Output AL = 4FH and

AH = 00H: Function was executed in an orderly manner

Remarks

The buffer that was passed must have the size which was indicated through the BX register, during the preceding call of the subfunction 04H/00H.

The various bits in the CX register indicate during the function call which components of the video status should be stored. See the subfunction 04H/00H.

Interrupt 10H, Function 4FH, Subfunction 04H/02H	VESA
Restore the video status of the Super VGA card	

After the storage of the video-status with the help of the subfunction 04H/01H, this status can be restored again through the call of this function.

Input

AH = 4FH

AL = 04H

DL = 01H

CX = Components of the video-status to be restored

ES:BX = FAR pointer to the segment address of the safety buffer

Output

AL = 4FH and

AH = 00H: Function was executed in an orderly manner

Remarks

The buffer which was passed, must have been loaded previously with information about the video status, through a call of the subfunction 04H/01H.

The various bits in the CX register indicate during the function call, which components of the video status should be restored. See the subfunction 04H/00H.

Interrupt 10H, Function 4FH, Subfunction 05H/00H	VESA
Determine access window in the video RAM	

This function is used to include a certain part of the video RAM in one of the two VESA access windows and thereby make it addressable for a program.

Input

AH = 4FH

AL = 05H

BH = 00H

BL = Access window (0 or 1)

DX = Start address

Output

AL = 4FH and

AH = 00H : Function was executed in an orderly manner

Remarks

The start address in the DX-Register should be viewed in relation to the granularity of the window, which can be determined by the call of the subfunction 01H.

The second access window can only be addressed with this function, if a preceding call of the subfunction 00H has indicated, that two access windows actually exist.

Interrupt 10H, Function 4FH, Subfunction 05H/01H	VESA
Query access window on the video RAM	

By using this function, the location of the access window in relation to the video RAM of the Super VGA card can be queried.

Input

AH = 4FH

AL = 05H

BH = 01H

BL = Access window (0 or 1)

Output

AL = 4FH and

AH = 00H: Function was executed in an orderly manner

DX = Starting address

Remarks

The start address in the DX register should be viewed in relation to the granularity of the window, which can be determined by the call of the subfunction 01H.

DOS API Interrupts And Functions

More than 100 functions can be accessed using Interrupt 21h, which are made available to a program by DOS and therefore are designated as the Application-Program-Interface (DOS-API).

These functions are described in this chapter, including a whole series of functions whose significance were never officially publicized by Microsoft and are therefore designated as undocumented. The "undocumented" functions which are described here, have been already used in many thousands of commercial applications and Microsoft can't afford to leave them out of the API in a future DOS version. If you find no official function to help you in some application, you may use one of these undocumented functions.

Interrupt 20H	DOS (Version 1.0 and above)
Terminate program	

Restores the three interrupt vectors whose contents were stored in the PSP before the program call, terminates the currently running program and returns control to MS-DOS. If the program redirected the vectors to its own routine, these vectors cannot be overwritten by another program. However, the terminating program releases the RAM it had occupied. Before turning control over to the calling program, this memory releases and all data buffers clear.

Input CS = Segment address of the PSP

Output No output

Remarks

COM programs automatically store the segment address of the PSP in the CS register. EXE programs require additional programming to load the segment address of the PSP into the CS register. Since the code and the PSP are stored in two separate segments, the address of the PSP must be loaded into the CS register. The code executes from another segment, which makes it impossible to call interrupt 32. To help overcome this problem, the value 0 and then the segment address of the PSP are pushed onto the stack. If a FAR RETURN command then executes, the program execution continues in the PSP segment at offset address 0. There a call for interrupt terminates the program.

For the first version of DOS, this interrupt is the usual method for ending a program. To terminate a program in DOS Version 2 and up, functions 31H or 4CH of DOS interrupt 21 H should be called instead.

Interrupt 21H, Function 00H	DOS (Version 1.0 and above)
Terminate program	

Terminates execution of the currently running program and returns control to the calling program. Before this happens, the three interrupt vectors, whose contents had been stored in the PSP before the call of the program, are restored. If the program redirects these vectors to its own routine, they cannot be overwritten by another program. However, the terminating program does release the RAM it had occupied. Before turning control over to the calling program, the function releases this memory and clears all buffers.

Input AH = 00H
CS = segment address of the PSP

Output No output

Remarks

COM programs automatically store, in the CS register, the segment address of the PSP. Since the code and the PSP are stored in two separate segments, you cannot execute this function from an EXE program.

Instead of this function, use either function 31H or 4CH of interrupt 21H for terminating a program.

Interrupt 21H, Function 01H	DOS (Version 1 and above)
Character input with echo	

Reads a character from the standard input device and displays it on the standard output device. When the function is called but a character doesn't exist, the function waits until a character is available. Since standard input and output can be redirected, this function is able to read a character from an input device other than the keyboard and send it to an output device other than the screen. The characters that are read may originate from other devices or from a file. If the character comes from a file, the input doesn't redirect to the keyboard once it reaches the end of the file. So, the function continues to try to read data from the file after it passes the end.

Input AH = 01H

Output AL = Character read

Remarks

If extended key codes are read, the function passes code 0 to the AL register. The function must be called again to read the actual code.

If the function encounters a **Ctrl C** character (ASCII code 3), it calls interrupt 23H.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 02H	DOS (Version 1 and above)
Character output	

Displays a character on the standard output device. Since this device can be redirected, the character can be displayed on another output device or sent to a file. This function doesn't test whether or not the storage medium (disk or hard disk) is already full. Therefore, it will continue to try to write characters to this file.

Input AH = 02H

DL = code of the character to be output

Output No output

Remarks

Control codes such as backspace, carriage return and linefeed are executed when the function sends characters to the screen. If the output is redirected to a file, control codes are stored as normal ASCII codes.

If the function encounters a **Ctrl C** character (ASCII code 3), it calls interrupt 23H.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 03H

DOS (Version 1 and above)

Read character auxiliary input

Reads a character from the serial port. Access defaults to the device with the designation COM1, unless a MODE command previously redirected serial access.

Input AH = 03H

Output AL = Character received

Remarks

Since the serial port has no internal buffer, it can receive characters faster than it can read them. The unread characters are then ignored.

Before calling this function, communication parameters (baud rate, number of stop bits, etc.) must be set using the MODE command. Otherwise DOS defaults to 2400 baud, one stop bit, no parity and a word length of 8 bits.

The BIOS functions called from interrupt 14H are a more efficient way to access the serial port. Since they also allow reading of the serial port status, these functions offer more flexibility than the DOS functions.

If the function encounters a **Ctrl C** character (ASCII code 3), it calls interrupt 23H.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 04H

DOS (Version 1 and above)

Auxiliary output

Sends a character to the serial port. Unless a MODE command previously redirected serial access, access defaults to the device with the designation COM1.

Input AH = 04H

DL = Character set for output

Output No output

Remarks

As soon as the receiving device sends a signal to the function indicating that it is ready to receive it, the function transmits the character. Control then returns to the calling program.

Before calling this function, communication parameters (baud rate, number of stop bits, etc.) must be set using the MODE command. Otherwise DOS defaults to 2400 baud, one stop bit, no parity and a word length of 8 bits.

The BIOS functions called from interrupt 14H are a more efficient way to access the serial port. Since they also allow reading of the serial port status, they offer more flexibility than the DOS functions.

If the function encounters a **Ctrl C** character (ASCII code 3), it calls interrupt 23H.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 05H**DOS (Version 1 and above)**

Character output to printer

Sends a character to the printer. Access defaults to the device with the designation LPT1 (identical to PRN), unless a MODE command previously redirected printer access.

Input

AH = 05H

DL = Character code to be printed

Output

No output

Remarks

The function transmits the character only when the printer signals that it is ready to receive it. Then control returns to the calling program.

If the function encounters a **Ctrl C** character (ASCII code 3), it calls interrupt 23H.

The BIOS functions called from interrupt 17H are more efficient for printer access. They offer more flexibility than the DOS printer functions for character output.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 06H**DOS (Version 1 and above)**

Direct console I/O

Reads characters from the standard input device and displays them on the standard output device. The read or written character isn't tested by the operating system (e.g., **Ctrl C** has no effect on the program). Since standard input and output can be redirected, this function can read a character from an input device other than the keyboard and sends it to an output device other than the screen. The characters read may originate from other devices or from a file. When writing characters, this function doesn't test whether the storage medium (disk or hard disk) is already full. Also, the calling program cannot determine whether all the characters have been read from an input file.

During character input, the function doesn't wait until a character is available. Instead, the function returns control to the calling program.

Input

AH = 06H

DL = 0-254: Send character code

DL = 255: Read a character

Output

Character output: No output

Character input: Zero flag=1: No character ready

Zero flag=0: Character read is in the AL register

Remarks

If extended key codes are read, the function passes code 0 to the AL register. The function must be called again to read the actual code.

ASCII code 255 (blank) cannot be displayed with this function because the function interprets ASCII code 255 as a command to input a character.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 07H**DOS (Version 1 and above)**

Unfiltered character input without echo

Reads a character from the standard input device without displaying the character on the standard output device. If a character doesn't exist when the function is called, the function waits until a character is available. The read character is not tested by the operating system (e.g., **Ctrl****C** has no effect on the program). Since standard input and output can be redirected, this function can read a character from an input device other than the keyboard. The characters that are read may originate from other devices or from a file. If the characters come from a file, the input doesn't redirect to the keyboard once it reaches the end of file. This causes the function to continue to try reading data from the file after it passes the end of file.

Input AH = 07H

Output AL =Character read

Remarks

If extended key codes are read, the function passes code 0 to the AL register. The function must be called again to read the actual code.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 08H**DOS (Version 1 and above)**

Character input without echo

Reads a character from the standard input device without displaying the character on the standard output device. If no character exists when the function is called, the function waits until a character is available.

Since standard input can be redirected, this function can read a character from an input device other than the keyboard. The characters read may originate from other devices or from a file. If the characters come from a file, the input doesn't redirect to the keyboard on reaching the end of file, so the function continues to try reading data from the file after it passes the end of file.

Input AH = 08H

Output AL =Character read

Remarks

If extended key codes are read, the function passes code 0 to the AL register. The function must be called again to read the actual code.

If the function encounters a **Ctrl****C** character (ASCII code 3), it calls interrupt 23H.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 09H**DOS (Version 1 and above)**

Output character string

Displays a character string on the standard output device. Since this device can be redirected, the character may be displayed on another output device or sent to a file. This function doesn't test whether or not the storage medium (disk or hard disk) is already full, and will continue to try to write the string to a file.

Input

AH = 09H

DS = String segment address

DX = String offset address

Output

No output

Remarks

The string must be stored in memory as a series of bytes which contain the ASCII codes of the characters to be output. A dollar sign character "\$" (ASCII code 36) indicates, to DOS, the end of the string.

Control codes, such as backspace, carriage return and linefeed, are executed within the string.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 0AH**DOS (Version 1 and above)**

Buffered input

Reads a number of characters from the standard input device and transmits the characters to a buffer. The input ends when the user presses the **Enter** key. The ASCII code of this key (13) is then placed in the buffer as the last character of the string.

Since standard input can be redirected, this function can read a character from an input device other than the keyboard. The characters read may originate either from other devices or from a file. If the characters come from a file, the input doesn't redirect to the keyboard on reaching the end of file, so the function continues to try reading data from the file after it passes the end.

Input

AH = 0AH

DS = Buffer segment address

DX = Buffer offset address

Output

No output

Remarks

The first byte of the buffer accepts the maximum number of characters (including the carriage return which ends the input) which can be read into the buffer, starting at memory location 2. In order to inform the function of the maximum number of characters it may read, this information must be entered, by the calling program, into the buffer before the function call.

After completion of the input, DOS places the number of characters read (excluding the carriage return) in memory location 1.

The buffer must be the number of the characters to be read plus 2 bytes.

When the input reaches the second to last memory location in the buffer, the computer beeps if you attempt to enter any character other than the **Enter** key (end of input).

Extended key codes occupy two bytes in the buffer. The first byte contains the code 0, and the second byte contains the extended key code.

If the function encounters a **Ctrl C** character (ASCII code 3), it calls interrupt 23H.

The **Backspace** and cursor keys let you edit the input without storing these keys in the buffer.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 0BH

DOS (Version 1 and above)

Get input status

Determines whether a character is available for reading from the standard input device.

Input AH = 0BH

Output AL = 0: No character available

AL = 255: One or more characters available for reading

Remarks

If the function encounters a **Ctrl C** character (ASCII code 3), it calls interrupt 23H.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 0CH

DOS (Version 1 and above)

Reset input buffer and then input

Clears the input buffer then calls one of the character input functions. Since all the character input functions get their characters from the standard input device and standard input may be redirected, this function only operates when the keyboard is the standard input device. In this case the characters could be entered before the function call but not read by a function. These existing characters are erased to ensure that the function call only reads characters which were inputted after its call.

Input AH = 0CH

AL = Function to be called during call of function 10

DS = Input buffer segment address

DX = Input buffer offset address

Output Functions 1, 6, 7 and 8: AL = Character to be read

Function 10: No output

Remarks

Functions 1, 6, 7, 8 and 10 can be passed to the function as calling functions.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 0DH	DOS (Version 1 and above)
Disk reset	

Sends all data stored in an internal DOS buffer to a block driver device (e.g., disk drive, hard disk). The open files (handles or FCBs) remain open.

Input AH = 0DH

Output No output

Remarks

Despite this function call, all open files must be closed in an orderly manner. Otherwise the current directory entry of the file may not update properly, which prevents access to new file data.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 0EH	DOS (Version 1 and above)
Select default disk drive	

Defines the the current default disk drive. Its designation appears as a prompt on the screen when the command interpreter expects input from the user. The drive indicated here will be used for all file access in which no special device was specified.

Input AH = 0EH

DL = Drive number

Output AL = Number of installed drives or volumes

Remarks

Drive A: has code number of 0, drive B: code number 1, etc.

Even if the PC has only one disk drive and one hard disk, the number of volumes in the AL register can be greater than two because the hard drive can be divided into multiple volumes. In addition, the PC can have one or more RAM disks as part of its configuration. For a PC with a single disk drive, you can only have two volumes because drive A: also simulates drive B:.

Unlike DOS Version 2, which permits 63 different device codes, DOS Version 3 permits 26 different devices (the letters A to Z). To keep compatibility between versions, limit your device access to a maximum of 26 devices.

BIOS interrupt 11H does a better job of reading the number of disk drives than this function.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 0FH	DOS (Version 1 and above)
Open file (FCB)	

Opens a file if one is available. After this function call executes successfully, the file can be read or written.

Input AH = 0FH

DS =FCB segment address of the file

DX = FCB offset address of the file

Output AL =0: File found and opened

AL =255: File not found

Remarks

Both normal and extended FCBs can be used.

If the file was found, DOS enters, into the FCB, the file size, the date and the time of its creation or last modification.

DOS sets the record length at 128 bytes. This record length can be changed in the FCB before opening a file. If you need a longer record length, the DTA must be moved (the original DTA is only 128 bytes long).

If random file access is performed, the random record field in the FCB must be set after the file opens successfully.

The file pointer points to the first byte of the file after the file opens.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 10H

DOS (Version 1.0 and above)

Close file (FCB)

Writes all data currently in the DOS buffer to the file and closes the file. In addition, the directory entry changes to reflect the new file size and the date and time of the most recent modification to the file.

Input AH = 10H

DS =FCB segment address of the file

DX = FCB offset address of the file

Output AL =0: File closed and directory entry revised

AL =255: File not found in directory

Remarks

Only open files can be closed.

For disk files, the disk which was in the drive when the function call occurred must also be the disk that contains the file. Otherwise, the function call writes an incorrect FAT and an incorrect directory to the disk, which makes the data that is already on the disk useless.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 11H

DOS (Version 1 and above)

Search for first match (FCB)

Searches for the first occurrence in the disk directory of the filename indicated in the FCB.

Input AH = 11H

DS = FCB segment address

DX = FCB offset address

Output AL = 0: File found

AL = 255: File not found

Remarks

The FCB passed to the function contains the drive specifier and the filename for which the function should search.

The filename can contain the wildcard "?" to search for a group of files.

The search is made only in the current directory of the indicated device.

If the function searches for a normal file, a normal FCB can pass the information to the function. However, if you wish to search for a file with special attributes (volume name, subdirectories, hidden files, etc.), extended FCBs must be used.

If a file was found, the DTA contains an FCB of the same type as the FCBs. This FCB in the DTA contains the found filename. For this reason, the DTA must always be large enough to accept either a normal or an extended FCB.

The DTA can be switched to its own buffer using function 1AH, to ensure that it is large enough to accept the FCB.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 12H	DOS (Version 1 and above)
Search for next match (FCB)	

Searches for additional occurrences in the disk directory of the filename indicated in the FCB, after the file was found by function 17 (see above).

Input AH = 12H

DS = FCB segment address

DX = FCB offset address

Output AL = 0: File found

AL = 255: File not found (no other files available)

Remarks

This function can only be called after calling function 11H.

The FCB passed to the function contains the drive specifier and the filename for which the function should search.

If another filename was found its name is recorded in the FCB at the beginning of the DTA.

The DTA can be switched with function 1AH to its own buffer to ensure that it is large enough to accept the FCB.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 13H**DOS (Version 1 and above)**

Delete file (FCB)

Erases one or more files in the current directory of the specified device.

Input

AH = 13H

DS = FCB segment address

DX = FCB offset address

Output

AL = 0: file(s) erased

AL = 255: No file(s) found, or file(s) assigned Read-Only attribute

Remarks

The FCB passed to the function contains both the device on which the files to be erased are located and the name of the file.

The filename can contain the wildcard "?" to erase a group of files.

Only files in the current directory of the indicated device may be erased.

If the function is used to delete a normal file, a normal FCB can pass the information to the function. However, if you want to delete a file with special attributes (volume name, subdirectories, hidden files, etc.), extended FCBs must be used.

Volumes may be deleted with this function, however, subdirectories may not.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 14H**DOS (Version 1 and above)**

Sequential read (FCB)

Reads the next sequential data block from a file.

Input

AH = 14H

DS = FCB segment address

DX = FCB offset address

Output

AL = 0: Block read

AL = 1: End of file reached

AL = 2: Segment wrap

AL = 3: Partial record read

Remarks

The function can only be called after the file was opened by the indicated FCB.

The DTA reads the block. If the DTA is not large enough, function 1AH must move the DTA into its own buffer.

The FCB records the size of the block and the corresponding number of bytes read.

Error 2 occurs when the DTA reaches the end of a segment and the block being read extends beyond the end of the segment.

Error 3 occurs when a partial block appears at the end of the file. The block is read in anyway and blank spaces bring the block up to the allocated block size.

After reading a block, the file pointer resets to the beginning of the next block so that the next function call automatically reads the next block.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 15H
DOS (Version 1 and above)

Sequential write (FCB)

Writes a sequential block to a file.

Input

AH = 15H

DS = FCB segment address

DX = FCB offset address

Output

AL = 0: Block written

AL = 1: Medium (disk/hard drive) full

AL = 2: Segment overflow

Remarks

The function can only be called after the file was opened by the indicated FCB.

The DTA writes the block it contains to the file. If the DTA is not large enough to hold the file, function 1AH must be used to move the DTA into its own buffer.

The FCB records the size of the block and the corresponding number of bytes written.

Error 2 occurs if the DTA reaches the end of a segment and the block being written extends beyond the end of the segment.

After writing a block, the file pointer resets to the beginning of the next block, so that the next function call automatically writes the next block.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 16H
DOS (Version 1 and above)

Create or truncate file (FCB)

Creates a new file, or dumps the contents of an existing file (file size=0 bytes). This function call allows other functions to read or write to the open file.

Input

AH = 16H

DS = FCB segment address

DX = FCB offset address

Output AL =0: File created or cleared

AL =255: File could not be created (e.g., directory full)

Remarks

The contents of an existing file called by this function are lost.

After calling this function, the file is already open; you don't need to open the file using function 0FH (see above).

If you open the file using an extended FCB, you can assign certain attributes to the file (e.g., volume name, hidden file, etc.).

You cannot create a subdirectory using this function.

After opening the file, the file pointer moves to the first byte of the file.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 17H

DOS (Version 1 and above)

Rename file (FCB)

Renames one or more files in the current directory of the specified device.

Input AH = 17H

DS =FCB segment address

DX = FCB offset address

Output AL =0: File(s) renamed

AL =255: No file found, or new filename matches old filename

Remarks

The FCB here is a special FCB, based on a normal FCB. The first 12 bytes contain the drive specifier and the name of the file to be renamed. However, this type of FCB has the new drive specifier and the new filename stored starting at memory location 10H. The drive specifier must be identical for both filenames.

The name of the file to be renamed can contain the wildcard "?", which renames several files. If the new filename contains the wildcard "?", the places in the filename and extension where a question mark appears in this parameter remain unchanged.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 18H

DOS (Version 1 and above)

Reserved for internal use

Interrupt 21H, Function 19H

DOS (Version 1 and above)

Get default disk drive

Returns the drive specifier of the default (current) disk drive.

Input AH = 19H

Output AL = Drive specifier

Remarks

This function identifies drive A as code 0, drive B as code 1, etc.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 1AH

DOS (Version 1 and above)

Set DTA address

Transfers the DTA (Disk Transfer Area) to another area of memory. The DTA acts as buffer memory for all FCB supported file accesses.

Input AH = 1AH

DS = New DTA segment address

DX = New DTA offset address

Output No output

Remarks

This function must be called if the existing DTA has insufficient memory to handle the transmitted data.

When the program starts, MS-DOS places the DTA at address 128 in the PSP. Since the program starts after address 255 of the PSP, it is 128 bytes long.

DOS does not test the length of the DTA. Instead it assumes that the DTA is large enough to accept the transmitted data. If this is not the case, a DOS function can overwrite the excess data.

DOS recognizes an error during various functions if the DTA is at the end of a segment and the data to be transmitted exceeds the end of the segment.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 1BH

DOS (Version 1 and above)

Get allocation information for default drive

Returns information about the format of the default drive.

Input AH = 1BH

Output AL = Number of sectors per cluster

DS = Media descriptor segment address

BX = Media descriptor offset address

DX = Number of clusters

Remarks

The media descriptor can return the following codes:

F8H: Hard disk

F9H: Disk drive: double-sided, 15 sectors per track (AT only)

FCH: Disk drive: single-sided, 9 sectors per track

FDH: Disk drive: double-sided, 9 sectors per track

FEH: Disk drive: single-sided, 8 sectors per track

FFH: Disk drive: double-sided, 8 sectors per track

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 1CH	DOS (Version 1 and above)
Get allocation information for specified drive	

Returns information about the format of the specified drive.

Input	<p>AH = 1CH</p> <p>DL = Drive specifier</p>
Output	<p>AL = Number of sectors per cluster</p> <p>DS = Media descriptor segment address</p> <p>BX = Media descriptor offset address</p> <p>DX = Number of clusters</p>

Remarks

This function identifies drive A as code 0, drive B as code 1, etc.

The media descriptor can return the following codes:

F8H: Hard drive

F9H: Disk drive: double-sided, 15 sectors per track (AT only)

FCH: Disk drive: single-sided, 9 sectors per track

FDH: Disk drive: double-sided, 9 sectors per track

FEH: Disk drive: single-sided, 8 sectors per track

FFH: Disk drive: double-sided, 8 sectors per track

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 1DH	DOS (Version 1 and above)
Reserved	

Interrupt 21H, Function 1EH	DOS (Version 1 and above)
Reserved	

Interrupt 21H, Function 1FH	DOS (Version 1 and above)
Get DPB pointer to current drive	

Gets the pointer to a DOS Parameter Block for the current disk drive.

Input AH = 1FH

Output AL = 000H: Operation successful
 AL = 0FFH: Error
 BX:DS = FAR pointer to DPB structure

Remarks

If the device indicated is a diskette drive, DOS accesses the drive to fill the DPB with data about the drive and the format of the diskette. For hard drives this information is already stored in the memory and no hardware access is required.

An error during the function call can only occur if an invalid drive code was indicated. If this occurs, error code 15 is returned in the AL register.

The construction of the DOS Parameter Block varies between DOS versions.

The contents of the AH, CX, DX, SI, DI, BP, CS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 20H	DOS (Version 1 and above)
Reserved	

Reads a specified file record into the DTA.

Input AH = 21H
 DS = FCB segment address
 DX = FCB offset address

Output AL = 0: Record read
 AL = 1: End of file reached
 AL = 2: Segment overflow
 AL = 3: Partial record read

Remarks

The function can only be called after the file was opened by the indicated FCB.

The record whose address is stored in the FCB starting at location 21H is read.

The DTA reads the record. If the DTA is not large enough, function 1AH must be called to move the DTA into its own buffer.

The FCB records the size of the record and the corresponding number of bytes read.

During the function call, the file pointer moves to the beginning of the record being read so that a subsequent call of a sequential read (function 14H-see above) reads the same record sequentially.

The record number does not increment following the function call, so a new call of this function would read the same record.

Error 2 occurs when the DTA reaches the end of a segment and the record being read extends beyond the end of the segment.

Error 3 occurs when a partial record appears at the end of the file. The record is read in anyway and blank spaces bring the record up to the allocated record size.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 22H

DOS (Version 1 and above)

Random write (FCB)

Writes data from memory to the specified record in a file.

Input

AH = 22H

DS = FCB segment address

DX = FCB offset address

Output

AL = 0: record was written

AL = 1: Medium (disk/hard drive) full

AL = 2: segment overflow

Remarks

The function can only be called after the file was opened by the indicated FCB.

The record whose address is stored in the FCB starting at location 21H is read.

The record is written from the DTA to the file. If the DTA is not large enough, function 1AH must move the DTA into its own buffer.

The FCB records the size of the record and the number of bytes read.

During the function call, the file pointer moves to the beginning of the record being read. This instructs subsequent calls of a sequential read (function 14H-see above) to read the same record sequentially.

The record number does not increment following the function call, so a new call of this function would read the same record.

Error 2 occurs when the DTA reaches the end of a segment and the record being written extends beyond the end of the segment.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 23H

DOS (Version 1 and above)

Get file size in records (FCB)

Determines the size of a file based on the number of records in that file.

Input

AH = 23H

DS = FCB segment address

DX = FCB offset address

Output AL = 0: Number of records found starting at FCB address 21H

AL = 255: File not found

Remarks

The FCB passed contains the drive specifier as well as the name and extension of the file to be examined.

Unlike the other FCB supported file accesses, the FCB requires the record size before the application can call this function.

A record size of 1 returns the size of the file in bytes.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 24H	DOS (Version 1 and above)
Set random record number	

Sets the record number in the FCB to the current position of the file pointer. Random access may begin at the point at which earlier sequential accesses left off.

Input AH = 24H
 DS = FCB segment address
 DX = FCB offset address

Output No output

Remarks

The function can only be called after the file was opened by the indicated FCB.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 25H	DOS (Version 1 and above)
Set interrupt vector	

Sets any interrupt vector to another routine.

Input AH = 25H
 AL = Interrupt number
 DS = New interrupt routine segment address
 DX = New interrupt routine offset address

Output No output

Remarks

Before calling this function, the old contents of the interrupt vector to be changed should be read and stored using function 35H. After the program terminates, the old contents of the interrupt vector should be restored.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 26H**DOS (Version 1 and above)**

Create PSP

Copies the PSP (program segment prefix) of the executing program to a specified address in memory.

Input

AH = 26H

DX = New PSP segment address

Output

No output

Remarks

The new PSP offset address is 0.

DOS Version 1 uses this function to execute other programs by creating a PSP, loading the program after this PSP and executing it.

For DOS Version 2 up, use the EXEC function 4BH to load and execute additional programs instead of this function.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 27H**DOS (Version 1 and above)**

Random block read (FCB)

Reads one or more sequentially stored records into memory.

Input

AH = 27H

CX = Number of records to be read

DS = FCB segment address

DX = FCB offset address

Output

AL = 0: Record read

CX = Number of records read

AL = 1: End of file reached

AL = 2: Segment overflow

AL = 3: Partial record read

Remarks

The function can only be called after the file was opened by the indicated FCB.

The starting record is the record whose address is stored in the FCB, starting at location 21H.

The record data passes to the DTA. If the DTA is not large enough, function 1AH must move the DTA into its own buffer.

The FCB records the size of the record and the corresponding number of bytes read.

After the function call, the file pointer moves to the end of the last record that was read so that it points to the next record (following the last record read).

Error 2 occurs when the DTA reaches the end of a segment and the record being read extends beyond the end of the segment.

Error 3 occurs when a partial record appears at the end of the file. The record is read in anyway and blank spaces bring the record up to the allocated record size.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 28H	DOS (Version 1.0 and above)
Random block write (FCB)	

Writes one or more records in sequence to the specified file.

Input

AH = 28H

CX = Number of records to be written

DS = FCB segment address

DX = FCB offset address

Output

AL = 0: Record written

CX = Number of records written

AL = 1: Medium (disk/hard drive) full

AL = 2: Segment overflow

Remarks

The function can only be called after the file was opened by the indicated FCB.

The starting record is the record whose address is stored in the FCB starting at location 21H.

The FCB records the size of the record and the corresponding number of bytes read.

The data is written from the DTA to the file. If the DTA is not large enough, function 1AH must move the DTA into its own buffer.

After the function call, the file pointer moves to the end of the last record written so that it points to the next record, which follows the last record written. The record number increments by the number of records written.

Error 2 occurs when the DTA reaches the end of a segment and the record being written extends beyond the end of the segment.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 29H	DOS (Version 1 and above)
Parse filename to FCB	

Transfers an ASCII format filename into the proper fields of an FCB. The filename can include a drive specifier, filename and file extension.

Input

AH = 29H

DS = Segment address of filename in memory

SI = Offset address of filename in memory

ES = FCB segment address

DI = FCB offset address

AL = Transmission parameters:

Bit 1 = 1: The drive specifier in the FCB changes only if the filename passed contains a drive specifier

0: The drive specifier changes anyway. If the filename passed contains no drive specifier, the the FCB defaults to 0 (current drive)

Bit 2 = 1: The filename in the FCB changes only if the filename parameter passed contains a filename

0: The filename changes. If the filename passed does not contain a filename, the filename in the FCB fills with spaces (ASCII code 32)

Bit 3 = 1: The file extension in FCB changes only if the filename passed contains an extension

0: The file extension in the FCB changes. If the filename passed has no extension, the extension field is padded with spaces (ASCII code 32)

Bits 4-8: Should contain the value 0

Output

AL = 0: The filename passed contains no wildcards

AL = 1: The filename passed contains wildcards

AL = 255: Invalid drive specifier

DS = Segment address of the first character after parsed filename

SI = Offset address of the first character after parsed filename

ES = FCB segment address

DI = FCB offset address

Remarks

The filename must end with an end character (ASCII code 0).

If the filename contains the wildcard "*", all corresponding fields in the FCB fill with the wildcard "?".

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 2AH**DOS (Version 1 and above)**

Get system date

Reads the current system date.

Input

AH = 2AH

Output

AL = Day of the week (0=Sunday, 1=Monday, etc.)

CX = Year

DH = Month

DL = Day

Remarks

DOS calls the clock driver to read the date.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 2BH	DOS (Version 1 and above)
Set system date	

Sets the current system date as returned by function 2AH (see above).

Input

AH = 2BH

CX = Year

DH = Month

DL = Day

Output

AL = 0: O.K.

AL = 255: Date incorrect

Remarks

The date passes to the clock driver.

If the PC does not have a realtime clock, the date remains in effect until the PC is switched off or rebooted.

If the date entry is incorrect, the PC retains the old date.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 2CH	DOS (Version 1 and above)
Get system time	

Gets the current system time.

Input

AH = 2CH

Output

CH = Hours

CL = Minutes

DH = Seconds

DL = Hundredths of a second

Remarks

DOS calls the clock driver to read the time.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 2DH**DOS (Version 1 and above)**

Set system time

Sets the current system time.

Input

AH = 2DH

CH = Hours

CL = Minutes

DH = Seconds

DL = hundredths of a second

Output

AL = 0: O.K.

AL = 255: Incorrect time

Remarks

The time passes to the clock driver.

If the PC does not have a realtime clock, the time remains in effect until the PC is switched off or rebooted.

If the time entry is incorrect, the PC retains the old time.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 2EH**DOS (Version 1 and above)**

Set verify flag

Sets the verify flag. This flag determines whether data should be verified after a write operation to a block driver for proper transmission.

Input

AH = 2EH

DL = 0

AL = 0: Don't verify data

AL = 1: Verify data

Output

No output

Remarks

This flag can be controlled at the user level with the VERIFY ON and VERIFY OFF commands.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 2FH	DOS (Version 1 and above)
Get DTA address	

Returns the address of the DTA (Data Transmission Area), which serves as a data buffer for all FCB supported file accesses.

Input AH = 2FH

Output ES = DTA segment address
 BX = DTA offset address

Remarks

This function determines the address of the DTA, but not the DTA's size.

After the start of a program, the DTA starts at memory location 128 of the PSP and has a length of 128 bytes.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 30H	DOS (Version 2 and above)
Get MS-DOS version number	

Returns the DOS version number.

Input AH = 30H

Output AL = Major version number (e.g., version 2.01=2)
 AH = Minor version number (e.g., version 3.01=01)

Remarks

The major (whole) version number represents the number preceding the decimal point. For example, the version number 3.3 returns the major version number 3.

The minor (fractional) version number represents the number following the decimal point. It is always given as two digits. For example, Version 2.1 returns the minor version number 10 (0AH).

If the AL register contains a value of 0, the program runs under DOS Version 1. DOS Version 1.0 cannot use this function.

The contents of the DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 31H	DOS (Version 2 and above)
Terminate and stay resident	

Terminates the currently executing program and returns control to the calling program. The current program remains in memory for later recall.

Input AH = 31H
 AL = Return code
 DX = Number of paragraphs to be reserved

Output No output

Remarks

The return code in the AL register indicates whether the program called by it correctly executes. The calling program can read this number by calling function 77 (4DH). This value can be tested from within a batch file using the ERRORLEVEL and IF commands.

The number of 16-byte paragraphs to be reserved indicates how many bytes, beginning with the PSP, cannot be released for other uses.

Memory blocks reserved by function 48H are not affected by the value in the DX register because they can only be released by calling function 49H.

Interrupt 21H, Function 32H

DOS (Version 1 and above)

Get DPB pointer for any drive

Gets the pointer to a DOS Parameter Block for any disk drive.

Input AH = 32H

DL = Drive code (0=current, 1=A, 2=B, etc.)

Output Carry flag=1: Error

Carry flag=0: O.K.

DS:BX = FAR pointer to the DPB

Remarks

If the device indicated is a diskette drive, DOS accesses the drive to fill the DPB with data about the drive and the format of the diskette. For hard drives this information is already stored in the memory and no hardware access is required.

An error during the function call can only occur if an invalid drive code was indicated. If this occurs, error code 15 is returned in the AL register.

The construction of the DOS Parameter Block varies between DOS versions.

The contents of the AH, CX, DX, SI, DI, BP, CS, SS and ES registers are not affected by this function.

Interrupt 2FH, Subfunction 1

DOS (Version 3.0 and above)

Send file to print spooler

Reads the **Ctrl Break** flag. This determines whether DOS should test for active **Ctrl C** or **Ctrl Break** keys on each function call, or on character input/output calls. **Ctrl C** and **Ctrl Break** trigger interrupt 23H.

Input AH = 33H

AL = 0

Output DL = 0: Test only during character input/output

DL = 1: Test on every function call

Remarks

Since the **Ctrl Break** flag is not part of the environment block of a program, it affects all programs which call the DOS character functions that test for **Ctrl C** or the **Break** key.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 33H, Subfunction 1	DOS (Version 2.0 and above)
Set Ctrl Break flag	

Sets and unsets the **Ctrl Break** flag. This determines whether DOS should test for the activation of the **Ctrl C** or **Ctrl Break** keys on each DOS function call or character input/output calls. **Ctrl C** and **Ctrl Break** trigger interrupt 23H.

Input

AH = 33H

AL = 1

DL = 0: Test only during character input/output

DL = 1: Test on every function call

Output No output

Remarks

Since the **Ctrl Break** flag is not part of the environment block of a program, it affects all programs which call the DOS character functions that test for **Ctrl C** or the **Break** key.

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 34H	DOS (Version 2 and above)
Get INDOS flag pointer	

Returns the address of the INDOS flag. This is especially important for minimizing re-entry problems in TSR programs.

Input AH = 34H

Output ES:BX = FAR pointer to the INDOS flag

Remarks

The INDOS flag is a byte, which counts the recursive calls made to interrupt 21H. It contains the value 0 as long as DOS is not active. This means that DOS functions can be called from a TSR program. If this flag contains a value larger than 0, DOS is presently active, where a value greater than 1 indicates a corresponding number of recursive calls.

If this occurs, DOS cannot be interrupted with a function call from a TSR program, because in the worst case a system crash occurs.

See Chapter 35 for more information on TSRs.

The contents of the CX, DX, SI, DI, BP, CS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 35H**DOS (Version 2 and above)**

Get interrupt vector

Returns the current contents of an interrupt vector and the address of the interrupt routine that belongs to it.

Input

AH = 35H

AL = Interrupt number

Output

ES = Interrupt routine segment address

BX = Interrupt routine offset address

Remarks

To ensure compatibility with future versions of DOS, instead of reading the vector's contents directly from the interrupt vector table, call this function for reading an interrupt vector.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 36H**DOS (Version 2 and above)**

Get free disk space

Returns information about the device (the block driver) from which the available memory space can be calculated.

Input

AH = 36H

DL = Device code

Output

AX = 65535: Device unavailable

AX < 65535: Number of sectors per cluster

BX = Number of available clusters

CX = Number of bytes per sector

DX = Total number of clusters on the device

Remarks

This function identifies drive A as code 0, drive B as code 1, etc.

The remaining memory on the medium can be computed from the number of bytes per sector multiplied by the number of sectors per cluster, multiplied by the number of free clusters.

The contents of the SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 37H, Subfunction 00H**DOS (Version 2 and above)**

Get code for command line switch

Gets the command line switch character used in program calls.

Input	AH = 37H
	AL = 00H
Output	DL = ASCII character code

Remarks

The default command line switch character is the division character (/). For this reason few programs check this code, and will not react to changes made to this code.

The contents of the BX, CX, DH, SI, DI, BP and segment registers are not affected by this function.

Interrupt 21H, Function 37H, Subfunction 01H	DOS (Version 2 and above)
Set code for command line switch	

Sets the command line switch character used in program calls.

Input	AH = 37H
	AL = 01H
	DL = ASCII character code
Output	Carry flag=0: Character accepted
	Carry flag=1: Error

Remarks

The equal sign (=), slash (/) and hyphen (-) characters are valid command line switch characters. Other characters are invalid and are rejected.

Since few programs make use of the subfunction 00H of this function and instead always assume the (/) character as code for the command-line-switch, the call of subfunction 01H has effect on few programs. This includes in every case the various internal and external DOS programs which in this regard behave admirably.

The contents of the BX, CX, DX, SI, DI, BP and segment registers are not affected by this function.

Interrupt 21H, Function 38H	DOS (Version 2 and above)
Get country	

Determines country-specific parameters, which are set in the CONFIG.SYS file using the DOS COUNTRY command.

Input	AH = 38H
	AL = 0
	DS = Buffer segment address
	DX = Buffer offset address

Output No output

Remarks

Before the function call, function 30H should be used to determine the DOS version. This can help the programmer compensate for differences between DOS versions during the call and return of this function.

The buffer must have at least 32 bytes allocated for recording the various country-specific parameters.

Following the function call, the individual bytes of this buffer contain the following information:

Bytes 0-1: Date format

0=USA: Month-day-year

1=Europe: day-month-year

2=Japan: Year-month-day

Byte 2: ASCII code of the currency symbol

Byte 3: 0

Byte 4: ASCII code of the thousand character (comma/period)

Byte 5: 0

Byte 6: ASCII code of decimal character (period/comma)

Byte 7: 0

Bytes 8-31: reserved

The contents of the processor registers and the flag registers are not affected by this function.

Interrupt 21H, Function 38H, Subfunction 00H

DOS (Version 3 and above)

Get country

Gets the country-specific parameters that are currently set.

Input

AH = 38H

DS = Buffer segment address

DX = Buffer offset address

AL = 0: read current country parameters

AL = 1-254: Country code parameters to be read

AL = 255: Country code parameters to be read placed in the BX register

Output

Carry flag=0: O.K.

Carry flag=1: Invalid country code

Remarks

Before the function call, function 30H should be used to determine the DOS version. This can help the programmer compensate for differences between DOS versions during the call and return of this function.

The buffer must have at least 32 bytes allocated for recording the various country specific parameters.

Following the function call, the individual bytes of this buffer contain the following information:

Bytes 0-1: Date format

0=USA: Month-day-year

1=Europe: Day-month-year

2=Japan: Year-month-day

Bytes 2-6: Currency indicator (string terminated by an end character)

Byte 7: ASCII code of the thousand character (comma/period)

Byte 8: 0

Byte 9: ASCII code of decimal character (period/comma)

Byte 10: 0

Byte 11: ASCII code of the date separation character

Byte 12: 0

Byte 13: ASCII code of the time separation character

Byte 14: 0

Byte 15: Currency format

bit 0=0: Currency symbol before the value

bit 0=1: Currency symbol after the value

bit 1=0: No spaces between value and currency symbol

bit 1=1: Space between value and currency symbol

Byte 16: Precision (number of decimal places)

Byte 17: Time format

bit 0=0: 12-hour clock

bit 0=1: 24-hour clock

Bytes 18-21: Address of character conversion routine (see below)

Bytes 22-33: reserved

Addresses 18 to 21 are the offset and segment addresses of a FAR procedure, which is used for accessing the country specific characters from the character set of the PC. The routine views the AL register's contents as the ASCII code of a lower case letter that should be converted to a capital letter. If a capital letter exists, it is retained in the AL register after the call. If the letter doesn't exist, the contents of the AL register remain unchanged. For example, the routine could be used to convert a lower case "a" into a capital "A".

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and the flag registers are not affected by this function.

Interrupt 21H, Function 38H, Subfunction 01H**DOS (Version 3 and above)**

Set country

Sets the current country-specific parameters. These parameters can be read using function 38H, subfunction 0. Previous versions of DOS required country-specific settings from the CONFIG.SYS file using the COUNTRY command. This function allows the user to set and change these parameters after booting.

Input

AH = 38H

DX = 65535

AL = 1-254: Number of the country

AL > 254: Look in BX for country number

BX = Number of the country (if AL > 254)

Output

Carry flag=0: O.K.

Carry flag=1: Invalid country code

Remarks

Before the function call, function 30H should be used to determine that this command exists.

This function only allows setting of the country code, for which DOS has preset parameters. These parameters cannot be changed from this function.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 39H**DOS (Version 2 and above)**

Create subdirectory

Creates a new subdirectory on the specified device.

Input

AH = 39H

DS = Subdirectory path segment address

DX = Subdirectory path offset address

Output

Carry flag=0: Subdirectory created

Carry flag=1: Error (AX = error code)

AX=3: Path not found

AX=5: Access denied

Remarks

The subdirectory path passed is an ASCII string which is terminated by an end character (ASCII code 0).

If the subdirectory path contains a drive specifier, the indicated device is accessed. Otherwise DOS creates the subdirectory on the current device.

An error can occur if any element of the path designation doesn't exist, a subdirectory already exists by that name, or the directory to be made is a subdirectory of the root directory and it is already filled.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 3AH**DOS (Version 2 and above)**

Delete subdirectory

Deletes a subdirectory from the specified drive.

Input

AH = 3AH

DS = Subdirectory path segment address

DX = Subdirectory path offset address

Output

Carry flag=0: Subdirectory deleted

Carry flag=1: Error (AX = error code)

AX=3: Path not found

AX=5: Access denied

AX=6: Directory to be deleted is the current directory

Remarks

The subdirectory path passed is an ASCII string which is terminated by an end character (ASCII code 0). If the subdirectory path contains a drive specifier, the indicated device is accessed. Otherwise DOS deletes the subdirectory from the current device.

An error can occur if any element of the path designation doesn't exist, the subdirectory is the current directory, or the directory to be deleted still contains files.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 3BH**DOS (Version 2 and above)**

Set current directory

Sets the current subdirectory for the device indicated.

Input

AH = 3BH

DS = Subdirectory path segment address

DX = Subdirectory path offset address

Output

Carry flag=0: Subdirectory set

Carry flag=1: Error (AX = error code)

AX=3: Path not found

Remarks

The subdirectory path passed is an ASCII string which is terminated by an end character (ASCII code 0).

If the subdirectory path contains a drive specifier, the indicated device is accessed. Otherwise DOS deletes the subdirectory from the current device.

An error can occur if any element of the path designation doesn't exist.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 3CH**DOS (Version 2 and above)**

Create or truncate file (handle)

Creates a new file, or dumps the contents of an existing file (file size=0 bytes). This function call allows other functions to read or write to the open file.

Input

AH = 3CH

CX = File attribute

Bit 0=1: File is read only

Bit 1=1: Hidden file

Bit 2=1: System file

DS = Filename segment address

DX = Filename offset address

Output

Carry flag=0: O.K. (AX = file handle)

Carry flag=1: Error (AX = error code)

AX=3: Path not found

AX=4: No available handle

AX=5: Access denied

Remarks

The various bits of the file attribute can be combined with each other.

The filename must be available as an ASCII string terminated by an end character (ASCII code 0). The filename parameter can contain a driver specifier, path, filename and extension. No wildcards are allowed. If you omit the drive specifier or path, DOS accesses the current drive or current directory.

An error can occur if any element of the path designation doesn't exist, if the file must be created in the root directory which is already full, or if a file with the same name already exists but cannot be cleared because it is write protected (bit 0 in the file attribute byte = 1).

If the function call executed successfully, all other handle functions can be called with this handle once the file opens.

The file pointer is set to the first byte of the file.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 3DH	DOS (Version 2 and above)
Open file (handle)	

Opens an existing file for access by other functions.

Input

AH = 3DH

AL = Access mode

Bits 0-2: Read/write access

000(b)=File is read only

001(b)=File can only be written

010(b)=File can be read and written

Bit 3: 0(b)

Bits 4-6: File sharing mode

000(b)=Only current program can access the file (FCB mode)

001(b)=Only the current program can access the file

010(b)=Another program can read but not write the file

011(b)=Another program can write but not read the file

100(b)=Another program can read and write the file

Bit 7: Handle flag

0=Child program of the current program can access file handle

1=Current program can access file handle only

DS = Filename segment address

DX = Filename offset address

Output

Carry flag=0: O.K. (AX = file handle)

Carry flag=1: Error (AX = error code)

AX=1:Missing file sharing software

AX=2: File not found

AX=3: Path not found or file doesn't exist

AX=4: No handle available

AX=5: Access denied

AX=12: Access mode not permitted

Remarks

The filename must be available as an ASCII string terminated by an end character (ASCII code 0). The filename parameter can contain a driver specifier, path, filename and extension. No wildcards are allowed. If you omit the drive specifier or path, DOS accesses the current drive or current directory.

If the function call executes successfully, all other handle functions can be called with this handle once the file opens.

The file pointer is set to the first byte of the file.

DOS Version 2 uses only bits 0 to 2 of the access mode. All other bits, even under Version 3, should be 0 to ensure proper execution of the call.

DOS Version 3 uses the file sharing mode in bits 4 to 6 of the access mode only if the file is on a mass storage device which is part of a network. These three bits decide if and how the file, while it is open using the current call, may be accessed by other programs from other PCs on the network.

Error 12 can occur only under DOS Version 3 and only within a network when the file is already opened by another program and if no other program can gain access to that file.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 3EH**DOS (Version 2 and above)**

Close file (handle)

Writes any data in the DOS buffers to a currently open file, then closes the file. If changes occur to the file, the new file size and the last date and time of modification are added to the directory.

Input

AH = 3EH

BX = Handle to be closed

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = error code)

AX=6: Unauthorized handle or file not opened

Remarks

Do not accidentally call this function with the numbers of the previous handle (the numbers 0 to 4) because the standard input device or standard output device may close. This would leave you unable to enter characters from the keyboard or display characters on the screen.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 3FH**DOS (Version 2 and above)**

Read file or device (handle)

Reads a certain number of characters by using a handle from a previously opened file or device and passes the characters to a buffer. The read operation starts at the current file pointer position.

Input

AH = 3FH

BX = File or device handle

CX = Number of bytes to be read

DS = Buffer segment address

DX = Buffer offset address

Output Carry flag=0: O.K. (AX = number of bytes read)

Carry flag=1: Error (AX = error code)

AX=5: Access denied

AX=6: Illegal handle or file not open

Remarks

Characters can be read from a file or from a device (e.g., the standard input device [keyboard], which has the handle 0).

When the carry flag resets after the function call but the AX register has the value 0, this means that the file pointer has already reached the end of the file before the function call. So, no files could be read.

When the carry flag resets after the function call but the contents of the AX register are smaller than the contents of the CX register before the function call, this means that the desired number of bytes wasn't read because the end of the file was reached.

After the function call, the file pointer follows the last byte read.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 40H

DOS (Version 2 and above)

Write to file or device (handle)

Writes a certain number of characters from a buffer to an open file or device by using a handle. The write operation begins at the file pointer's current position.

Input AH = 40H

BX = File or device handle

CX = Number of bytes to be written

DS = Buffer segment address

DX = Buffer offset address

Output Carry flag=0: O.K. (AX = number of bytes written)

Carry flag=1: Error (AX = error code)

AX=5: Access denied

AX=6: Illegal handle or file not open

Remarks

Characters can be written to a file or to a device (e.g., the standard output device [screen], which has the handle 1).

When the carry flag resets after the function call but the AX register has the value 0, this means that the file pointer has already reached the end of the file before the function call. Therefore no files could be written.

When the carry flag resets after the function call but the contents of the AX register are smaller than the contents of the CX register before the function call, this means that the desired number of bytes were not written because the end of file was reached.

After the function call, the file pointer follows the last byte written.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 41H

DOS (Version 2 and above)

Delete file (handle)

Deletes the filename passed to the function. Through the call of this function, a file is erased and its name is passed to the function.

Input

AH = 41H

DS = Filename segment address

DX = Filename offset address

Carry flag=0: O.K.

Carry flag=1: Error (AX = error code)

AX=2: File not found

AX=5: Access denied

Remarks

The filename must be available as an ASCII string terminated by an end character (ASCII code 0). The filename parameter can contain a drive specifier, path, filename and extension. No wildcards are allowed. If you omit the drive specifier or path, DOS accesses the current drive or current directory.

An error occurs when any element of the path designation doesn't exist or when the file has the attribute Read Only and therefore can not be written to or deleted. This attribute can be changed by using function 43H.

You cannot delete subdirectories or volume names with this function.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 42H

DOS (Version 2 and above)

Move file pointer (handle)

Moves the file pointer of a previously opened file by using its handle. This allows random access because the individual records don't have to be read in sequence. The new file pointer position is given as an offset from the current position, either from the beginning of the file or from the end of the file. The offset itself is indicated as a 32-bit number.

Input

AH = 42H

AL = Offset code

AL=0: Offset is relative to the beginning of the file

AL=1: Offset is relative to the current position of the file pointer

AL=2: Offset is relative to the end of the file

BX = Handle
 CX = High word of the offset
 DX = Low word of the offset

Output Carry flag=0: O.K.

DX=High word of the file pointer
 AX=Low word of the file pointer
 Carry flag=1: Error (AX = error code)
 AX=1: Illegal offset code
 AX=6: Illegal handle or File not open

Remarks

If offset codes 1 and 2 are accessed, negative offsets may be used to move the file pointer backwards or to place the pointer at the beginning of the file. It's possible to set the file pointer before the end of the file, which causes an error during the next read or write access to the file.

The position of the file pointer passed after the function call is always relative to the beginning of the file. The offset code used during the function call is independent of this file pointer position.

Passing offset code 2 and offset 0 returns the size of the file. This action moves the file pointer to the last byte of the file and the pointer's position returns to the calling program after the function call.

The contents of the BX, CX, , SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 43H, Subfunction 0

DOS (Version 2 and above)

Get file attributes

Determines file attributes.

Input AH = 43H
 AL = 0
 DS = Filename segment address
 DX = Filename offset address

Output Carry flag = 0: O.K. (CX = file attribute)
 Bit 0=1: File can be read but not written
 Bit 1=1: File hidden (not displayed on DIR)
 Bit 2=1: File is a system file
 Bit 3=1: File is the volume name
 Bit 4=1: File is a subdirectory
 Bit 5=1: File was changed since the last date/time

Carry flag = 1: Error (AX = error code)

AX=1: Unknown function code

AX=2: File not found

AX=3: Path not found

Remarks

The filename must be available as an ASCII string terminated by an end character (ASCII code 0). The filename parameter can contain a driver specifier, path, filename and extension. No wildcards are allowed. If you omit the drive specifier or path, DOS accesses the current drive or current directory.

An error occurs when any element of the path designation or the file does not exist. The contents of the BX, CX, , SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 43H, Subfunction 1

DOS (Version 2 and above)

Set file attributes

Sets the file attributes.

Input

AH = 43H

AL = 1

CX = File attributes

Bit 0=1: File can be read but not written

Bit 1=1: File hidden (not displayed on DIR)

Bit 2=1: File is a system file

Bit 3=0

Bit 4=0

Bit 5=1: File was changed since the last date/time

DS = Filename segment address

DX = Filename offset address

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = error code)

AX=1: Unknown function code

AX=2: File not found

AX=3: Path not found

AX=5: Attribute cannot be changed

Remarks

The filename must be available as an ASCII string terminated by an end character (ASCII code 0). The filename parameter can contain a driver specifier, path, filename and extension. No wildcards are allowed. If you omit the drive specifier or path, DOS accesses the current drive or current directory.

An error occurs when any element of the path designation or the file does not exist.

Neither subdirectories nor volume names can be accessed with this function. For this reason bits 3 and 4 of the file attribute must be 0 during the function call. If you attempt to access a subdirectory or a volume name anyway, the function returns error code 5.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 1	DOS (Version 2.0 and above)
IOCTL: Set device information	

Permits access of a character driver's device attribute.

Input

AH = 44H

AL = 0

BX = Handle

Carry flag=0: O.K. (DX = device attribute)

Bit 14=1: Processes control characters through IOCTL

Bit 7=1: Character driver

Bit 5=0: Cooked mode operation

Bit 5=1: Raw mode operation

Bit 3=1: Clock driver operation

Bit 2=1: NUL driver operation

Bit 1=1: Console output driver (screen)

Bit 0=1: Console input driver (keyboard)

Carry flag=1: Error (AX = error code)

AX=1: Unknown function code

AX=6: Handle not opened or does not exist

Remarks

A handle is passed (not the name of the addressed character driver which must be connected with this driver). This can be one of the five pre-assigned handles (0 to 4). A handle could have been previously opened for a certain device with the help of the Open function (function 3DH), and then passed to the function. For example, since the standard input and output devices (handles 0 and 1) can be redirected, this method assures that the indicated device is accessed.

If bit 7 in the device attribute is unequal to 1, the driver addressed is not a character driver and the significance of the individual bits in the device attribute disagrees with those of the device driver. The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 1**DOS** (*Version 2.0 and above*)

IOCTL: Set device information

Sets the character device attributes.

Input

AH = 44H

AL = 1

BX = Handle

CX = Number of bytes written

DX = Device attributes

Bit 14=1: Processes control characters through IOCTL using subfunctions 2 and 3

Bit 7=1: Character driver

Bit 5=0: Cooked mode operation

Bit 5=1: Raw mode operation

Bit 3=1: Clock driver operation

Bit 2=1: NUL driver operation

Bit 1=1: Console output driver (screen)

Bit 0=1: Console input driver (keyboard)

Carry flag=0: O.K.

Output

Carry flag=1: Error (AX = Error code)

AX=1: Unknown function code

AX=6: handle not opened or handle does not exist

Remarks

A handle is passed but it is not the name of the addressed character device, which must be connected with this device. This can be one of the five pre-assigned handles (0 to 4). A handle could have previously been opened, with the Open function, for a certain device and then passed to the function. For example, since the standard input and output devices (handles 0 and 1) can be redirected, this method assures that the indicated device is accessed.

To change various device attribute bits with this function, use subfunction 0 to read the device attributes first. Then this subfunction can reset the device attribute bits in the device driver.

If bit 7 in the device attribute is unequal to 1, the driver addressed is not a character driver. The meanings of the individual bits in the device attribute disagree with those in the device driver.

This function is especially useful for switching between cooked mode and raw mode within a character driver (bit 5).

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 2	DOS (Version 2.0 and above)
IOCTL: Read data from character device	

Reads data from a character device. This function defines the number of bytes of data to read from the buffer, which contains the data taken from the character device.

Input	AH = 44H
	AL = 2
	BX = Handle
	CX = Number of bytes to be read
	DS = Buffer segment address
	DX = Buffer offset address
Output	Carry flag=0: O.K. (AX = Number of bytes sent)
	Carry flag=1: Error (AX = Error code)
	AX=1: Unknown function code
	AX=6: Handle not opened or does not exist

Remarks

A handle is passed, but it is not the name of the addressed character device which must be connected with this device. This can be one of the five pre-assigned handles (0 to 4). A handle could have previously been opened with the Open function (function number 3DH) for a certain device, then passed to the function. For example, since the standard input and output devices (handles 0 and 1) can be redirected, this method assures that the indicated device is accessed.

An error always occurs if the handle passed is connected with a block driver instead of a character driver.

The driver defines the data type and structure.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 3	DOS (Version 2.0 and above)
IOCTL: Send data to character device	

Sends data from an application program directly to a character device. The calling function defines the number of bytes to be transferred from a buffer to the device.

Input	AH = 44H
	AL = 3
	BX = Handle
	CX = Number of bytes to be transmitted
	DS = Buffer segment address
	DX = Buffer offset address

Output Carry flag=0: O.K.

AX=Number of bytes sent

Carry flag=1: Error (AX = Error code)

AX=1: Unknown function code

AX=6: Handle not opened or does not exist

Remarks

A handle is passed, but it is not the name of the addressed character device which must be connected with this device. This can be one of the five pre-assigned handles (0 to 4). A handle could have previously been opened with the Open function (function number 61) for a certain device, then passed to the function. For example, since the standard input and output devices (handles 0 and 1) can be redirected, this method assures that the indicated device is accessed.

An error always occurs if the handle passed is connected with a block driver instead of a character driver.

The driver defines the data type and structure.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 4

DOS (Version 2.0 and above)

IOCTL: Read data from block device

Reads data for an application directly from a block device. The calling function defines the number of bytes to be copied by the device into a buffer.

Input

AH = 44H

AL = 4

BX = Device designation

CX = Number of bytes to be read

DS = Buffer segment address

DX = Buffer offset address

Carry flag=0: O.K.

AX = Number of bytes sent

Output

Carry flag=1: Error (AX = Error code)

AX=1: Unknown function code

AX=15: Unknown device

Remarks

Instead of defining the device driver, the device designation parameter defines the device from which data will be received. Code 0 represents device A:, 1 represents device B:, etc.

The driver defines the data type and structure. The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 4	DOS (Version 6.0 and above)
IOCTL / DoubleSpace: Write internal caches	

Instructs DoubleSpace to write the contents of its internal cache buffers for cluster data, BitFAT and MDFAT to disk.

Input

AH = 44H

AL = 04H

BX = Device designation (0 = A)

CX = Number of bytes to be read

DS:DX = FAR pointers to buffer (see below)

Output

Carry flag = 0: OK

Carry flag = 1: Error

Remarks

In BX you can enter the device designation of any compressed drive.

The buffer referred to by DS:DX upon calling the function must contain the following:

Offset	Contents	TaskType	
+00H	'DM'	For identification of Microsoft DoubleSpace	1 word
+02H	'F'	For "Flush" instruction	1 byte
+03H	0	Receives error code following call	1 word
+05H	0, 0, 0, 0, 0	Five fill bytes	5 bytes

Length: 10 bytes

Following the function call, the word at offset address 03H of the buffer contains the character combination "OK" if the function was carried out successfully.

Interrupt 21H, Function 44H, Subfunction 4	DOS (Version 6.0 and above)
IOCTL / DoubleSpace: Write internal caches and invalidate	

Instructs DoubleSpace to write the contents of its internal cache buffers for cluster data, BitFAT and MDFAT to disk, then declare the contents of these buffers invalid.

Input

AH = 44H

AL = 04H

BX = Device designation (0 = A)

CX = Number of bytes to be read

DS:DX = FAR pointer to buffer (see below)

Carry flag = 0: OK

Carry flag = 1: Error

Remarks

In BX you can enter the device designation of any compressed drive.

The buffer referred to by DS:DX upon calling the function must contain the following:

Offset	Contents	TaskType	
+00H	'DM'	For identification of Microsoft DoubleSpace	1 word
+02H	'I'	For "Invalidate" instruction	1 byte
+03H	0	Receives error code following call	1 word
+05H	0, 0, 0, 0, 0	Five fill bytes	5 bytes

Length: 10 bytes

Following the function call, the word at offset address 03H of the buffer contains the character combination "OK" if the function was carried out successfully.

Interrupt 21H, Function 44H, Subfunction 5

DOS (Version 2.0 and above)

IOCTL: Send data to block device

Sends data from an application program directly to a character device. The calling function defines the number of bytes to be transferred from a buffer to the device.

Input

AH = 44H
 AL = 5
 BX = Device designation
 CX = Number of bytes to be sent
 DS = Buffer segment address
 DX = Buffer offset address

Output

Carry flag=0: O.K.
 AX=Number of bytes sent
 Carry flag=1: Error (AX = Error code)
 AX=1: Unknown function code
 AX=15: Unknown device

Remarks

Instead of defining the device driver, the device designation parameter defines the device from which data will be received. Code 0 represents device A:, 1 represents device B:, etc.

The driver defines the data type and structure. The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 6	DOS (Version 2.0 and above)
IOCTL: Read input status	

Determines whether a device driver can transmit data to an application program.

Input

AH = 44H

AL = 6

BX = Handle

Output

Carry flag=0: O.K. (AX = Input status)

AX=0: Driver not ready

AX=255: Driver ready

Carry flag=1: Error (AX = Error code)

AX=1: Unknown function code

AX=5: Access denied

Remarks

The handle passed can refer to either a character driver or a file.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 7	DOS (Version 2.0 and above)
IOCTL: Read output status	

Determines whether a device driver can receive data from an application program.

Input

AH = 44H

AL = 7

BX = Handle

Output

Carry flag=0: O.K. (AX = Output status)

AX=0: Driver not ready

AX=255: Driver ready

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function number

AX=5: Access denied

Remarks

The handle passed can refer to either a character driver or a file.

If the handle refers to a file, the block device driver signals its readiness to receive data, even if the medium containing the file is full and no additional data can be appended to the end of the file.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 8	DOS (Version 3.0 and above)
IOCTL: Test for changeable block device	

Determines whether the block device medium (e.g., disk, hard drive, etc.) can be changed.

Input	AH = 44H
	AL = 8
	BL = Device designation
Output	Carry flag=0: O.K. (AX=status code)
	AX=0: Medium changeable
	AX=1: Medium unchangeable
	Carry flag=1: Error (AX = Error code)
	AX=1: Invalid function number
	AX=15: Invalid drive number

Remarks

The device designation parameter defines the device being addressed instead of the device driver. Code 0 represents device A:, 1 represents device B:, etc.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 9	DOS (Version 3.1 and above)
IOCTL: Test for local or remote drive	

Determines whether a drive (block device) is local (part of the PC making the inquiry) or remote (part of another PC in a network).

Input	AH = 44H
	AL = 9
	BL = Device designation
Output	Carry flag=0: O.K.
	DX = device attribute
	Bit 12=0: Local
	Bit 12=1: Remote
	Carry flag=1: Error (AX = Error code)

AX=1: Invalid function number

AX=15: Invalid drive specification

Remarks

You can access this subfunction only if networking software has previously been installed.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 0AH	DOS (Version 3.1 and above)
IOCTL: Test for local or remote handle	

Determines whether a file associated with this handle is local (part of the PC making the inquiry) or remote (part of another PC in a network).

Input

AH = 44H

AL = 0AH

BX = Handle

DX = IOCTL code

Bit 15 = 0: Local

Bit 15 = 1: Remote

Output

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function number

AX=6: Handle not opened or does not exist

Remarks

You can access this subfunction only if networking software has been installed.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 0BH	DOS (Version 3.0 and above)
IOCTL: Change retry count	

Sets the variables that specify the number of attempts at file access. One PC within a network may try to access a file that is already being accessed by another PC. The PC attempting access repeats the file access procedure the number of times and the number of waiting periods defined by these variables.

Input

AH = 44H

AL = 0BH

BX = Number of attempts

CX = Waiting time between attempts

Output Carry flag=0: O.K.
 Carry flag=1: Error (AX = Error code)
 AX=1: Invalid function number

Remarks

You can only access this subfunction if networking software has previously been installed.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 0CH	DOS (Version 3.3 and above)
IOCTL: Character driver access	

Permits character driver access to code pages. Since code pages are only supported by certain character drivers, these functions are available to these drivers only.

Input AH = 44H
 AL = 0CH
 BX = Handle
 CH = Device designation
 CL = Device subfunction (see below)
 DS:DX = FAR pointer to buffer and other information

Output Carry flag=0: O.K.
 Carry flag=1: Error (AX = Error code)

Remarks

The CH register specifies the device designation, using one of the following numbers:

- 1 = Serial device (COM1, COM2, COM3 or COM4)
- 2 = Display device (CON)
- 5 = Parallel device (LPT1, LPT2 or LPT3)
- 0 = None of the above

The following subfunctions can be addressed through the CL register:

- 45H = Set iteration count for printer driver (DOS 3.3 and up)
- 4AH = Select code page (DOS 3.3 and up)
- 4CH = Start code page initialization (DOS 3.3 and up)
- 4DH = End code page initialization (DOS 3.3 and up)
- 5FH = Set display mode (DOS 4.0 and up)
- 65H = Get iteration count for printer driver (DOS 3.3 and up)

6AH = Query current code page (DOS 3.3 and up)

6BH = Query code page initialization list (DOS 3.3 and up)

7FH = Get display mode (DOS 4.0 and up)

Calling this function requires the following steps:

1. Call function 60H to query the current parameter. Verify and store the current parameter for later recall.
2. Call function 40H to set the new parameter.
3. Execute function 44H, subfunction 0CH.
4. Call function 40H to restore the original parameter.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 0DH	DOS (Version 3.2 and above)
IOCTL: Block driver access	

Permits block driver access to code pages. Since code pages are only supported by certain character drivers, these functions are available to these drivers only.

Input

AH = 44H

AL = 0DH

BL = Device designation

CH = 08H

CL = Device subfunction (see below)

DS:DX = FAR pointer to buffer and other information

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

Remarks

The device designation parameter defines the device being addressed instead of the device driver. Code 0 represents device A:, 1 represents B:, etc.

The following subfunctions can be addressed through the CL register:

40H = Set device parameters (DOS 3.2 and up)

41H = Write track on logical drive (DOS 3.2 and up)

42H = Format track on logical drive (DOS 3.2 and up)

46H = Set media ID (DOS 4.0 and up)

60H = Determine device parameters (DOS 3.2 and up)

61H = Read track on logical drive (DOS 3.2 and up)

62H = Verify track on logical drive (DOS 3.2 and up)

66H = Get media ID (DOS 3.2 and up)

68H = Sense media type (DOS 5.0 and up)

Calling this function requires the following steps:

1. Call function 60H to query the current parameter. Verify and store the current parameter for later recall.
2. Call function 40H to set the new parameter.
3. Execute function 44H, subfunction 0CH.
4. Call function 40H to restore the original parameter.
5. The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 0EH	DOS (Version 3.2 and above)
IOCTL: Get logical drive map	

Determines whether multiple logical drives are assigned to one block device.

Input

AH = 44H

AL = 0EH

BL = Device code (0=current, 1=A, 2=B, etc.)

Output

Carry flag=0: O.K.

AL = 0 : Device is a logical device

AL =>1 : Number of logical device (1=A:, 2=B:, etc.)

Carry flag=1: Error (AX = error code)

Remarks

This function is of interest mainly in relation to system with only one diskette drive, because this drive can be addressed as A and B. With the help of this function it can be determined if the drive was last addressed as either A or B. If the drive is addressed through this query, a DOS message for insertion of a diskette may be avoided during the access to the drive.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 0FH	DOS (Version 3.2 and above)
IOCTL: Set logical drive map	

With this function the logical drive designation can be determined, through which a device is addressed during the next access.

Input

AH = 44H

AL = 0FH

BL = Device code (0=current, 1=A, 2=B, etc.)

Output Carry-Flag=0: o.k., in this case

AL = Device code, which the drive is addressed during the next access.

Carry-Flag=1: Error, in this case AX = Error-Code

Remarks

This function is interesting mainly in relation to systems with only one diskette drive, because this drive can be addressed as either A or B. In this case, the diskette drive can be switched with the help of this function from A to B. Since the function works alternately, a subsequent call will switch a drive to A again.

The contents of the BX, CX, DX, SI, DI, BP and the segment Registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 10H

DOS (Version 5.0 and above)

IOCTL: Query IOCTL handle

With this function the availability of an IOCTL-function, which is not available in version 3.2, can be queried. This function is only available after version 5.0.

Input

AH = 44H

AL = 10H

BX = Handle, whose device should be considered during the call of the IOCTL function.

CL = Number of the desired IOCTL function

Output

Carry-Flag=0: Function is available in relation to the indicated Handle

Carry-Flag=1: Error, function not available in relation to the Handle

AX = 1 (Function not available)

Remarks

While the availability of a certain IOCTL function can be tested with the help of this function only in relation to the device which hides behind a Handle, the IOCTL function 11H makes a test in relation to a certain device possible.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 44H, Subfunction 11H

DOS (Version 5.0 and above)

IOCTL: IOCTL device

After version 5.0, this function permits the query of the availability of this IOCTL function, before the call of an IOCTL function which was not included in the functions of Version 3.2.

Input

AH = 44H

AL = 11H

BX = Device-Number (0=current, 1=A, 2=B, etc.)

CL = Number of the desired IOCTL function

Output Carry-Flag=0: Function is available in relation to the indicated device

Carry-Flag=1: Error, function is not available in relation to the device

AX = 1 (Function not available)

Remarks

While the availability of a certain IOCTL function can be tested with the help of this function only in relation to the device which hides behind a Handle, the IOCTL function 10H makes a test possible in relation to a certain Handle and the device which is hidden behind it.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 45H**DOS (Version 2.0 and above)**

Duplicate handle

Creates a duplicate of the handle passed. This duplicate handle interfaces with the same file or device as the first handle. If the first handle refers to a file, the value of the first handler's file pointer joins with the file pointer of the duplicate handle.

Input AH = 45H

BX = Handle

Output Carry flag=0: O.K. (AX = New handle)

Carry flag=1: Error (AX = Error code)

AX=4: No additional handle available

AX=6: Handle not opened or does not exist

Remarks

Without having to close the file, this function updates a file directory entry after its modification. A file can be closed using function 62 (3EH).

If the file pointer of one of the two handles changes position due to the call of a read or write function, the other file pointer also changes automatically.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 45H**DOS (Version 2.0 and above)**

IOCTL: Duplicate handle

Refers a second file handle to the save device or file as the first file handle. The second handle's file pointer also contains the same value as the first handle's file pointer.

Input AH = 46H

BX = First handle

CX = Second handle

Output Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=4: No additional handle available

AX=6: Handle not opened or does not exist

Remarks

If the function call connects the second handle to an open file, the file closes before the forced duplication.

If the file pointer of one of the handles changes position due to the call of a read or write function, the other file pointer also changes automatically.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 47H

DOS (Version 2.0 and above)

Get current directory

Gets an ASCII string listing the complete path designation of the current directory of the indicated device. This string passes to the specified buffer.

Input

AH = 47H

DL = Device designation

DS = Buffer segment address

SI = Buffer offset address

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX=Error code)

AX=15: Invalid drive specification

Remarks

The device designation parameter defines the device being addressed instead of the device driver. Code 0 represents the current device, 1 represents device A:, etc.

The path description in the buffer terminates with an end character (ASCII code 0). This description has no drive specifier or \ character (root directory specifier). If the root directory is the current directory, the end character becomes the first character in the buffer.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 48H

DOS (Version 2.0 and above)

Allocate memory

Reserves an area of memory for program use.

Input

AH = 48H

BX = Number of paragraphs to be reserved

Output Carry flag=0: O.K.

 AX=Memory area segment address)

 Carry flag=1: Error (AX = Error code)

 AX=7: Memory control block destroyed

 AX=8: Insufficient memory

 BX = Number of paragraphs available

Remarks

A paragraph consists of 16 bytes.

If memory allocation was successfully executed, the allocated range begins at address AX:0000.

This function always fails when executed from within a COM program because the PC assigns the total amount of free memory to a COM program when it executes.

The contents of the CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 49H**DOS (Version 2.0 and above)**

Release memory

Releases memory previously allocated by function 72 (49H-see above) for any purpose.

Input AH = 49H

 ES = Memory area segment address

Output Carry flag=0: O.K.

 Carry flag=1: Error (AX = Error code)

 AX=7: Memory control block destroyed

 AX=9: Incorrect memory area passed in ES

Remarks

Since DOS knows the size of the memory area to be released, no parameter exists for passing memory size.

If the wrong segment address appears in the ES register during the function call, memory assigned to another program can be released. This can lead to a system crash or other consequences.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 4AH**DOS (Version 2.0 and above)**

Modify memory allocation

Changes the size of a memory area previously reserved using function 72 (3FH-see above).

Input

AH = 4AH

BX = New memory area size in paragraphs

ES = Memory area segment address

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=7: Memory control block destroyed

AX=8: Insufficient memory

BX = Number of paragraphs available

Remarks

A paragraph has 16 bytes.

If the wrong segment address appears in the ES register during the function call, memory assigned to another program can be released. This can lead to a system crash or other consequences.

Since the PC assigns the total amount of free memory to a COM program when it executes, this function call always fails when executed from within a COM program.

COM programs should use this function to release all unnecessary memory since all RAM becomes part of a COM program. This is especially important before calling the EXEC function (function number 75 (4BH)).

The contents of the CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 4BH, Subfunction 0

DOS (Version 2.0 and above)

Execute program

Executes another program from within a program and continues execution of the original program after the called program finishes its run. The function requires the name of the program to be executed and the address of a parameter block, which contains information that is important to the function.

Input

AH = 4BH

AL = 0

ES = Parameter block segment address

BX = Parameter block offset address

DS = Program name segment address

DX = Program name offset address

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function number

AX=2: Path or program not found

AX=5: Access denied

AX=8: Insufficient memory

AX=10: Wrong environment block

AX=11: Incorrect format

Remarks

The directory name passed is an ASCII string which is terminated by an end character (ASCII code 0). It can contain a path designation and drive specifier. No wildcards are allowed. If no drive specifier or path designation exists, the function accesses the current drive or directory.

Only EXE or COM programs can be executed. To execute a batch file, the command processor (COMMAND.COM) must be called using the /c parameter followed by the name of the batch file.

The parameter block must have the following format:

Bytes 0-1:	Environment block segment address
Bytes 2-3:	Command parameter offset address
Bytes 4-5:	Command parameter segment address
Bytes 6-7:	First FCB offset address
Bytes 8-9:	First FCB segment address
Bytes 10-11:	Second FCB offset address
Bytes 12-13:	Second FCB segment address

If the segment address of the environment block is a 0, the called program has the same environment block as the calling program.

The command parameters must be stored so that the parameter string begins with a byte representing the number of characters in the command line. Next follow the individual ASCII characters, which are terminated by a carriage return (ASCII code 13) (this carriage return is not counted as a character).

The first FCB passed is copied to the PSP of the called program starting at address 5CH. The second FCB passed is copied to the PSP of the called program starting at address 6CH. If the called program does not obtain information from the two FCBs, any desired value can be entered into the FCB fields at the parameter block.

After the call of this function, all registers are destroyed except the CS and IP registers. For later recall, save their contents before the function call.

The program called should have all the handles available to the calling program.

Interrupt 21H, Function 4BH, Subfunction 3

DOS (Version 2.0 and above)

Execute overlay

Loads a second program into memory as an overlay without automatically executing the second program.

Input

AH = 4BH

AL = 3

ES = Parameter block segment address

BX = Parameter block offset address
 DS = Program name segment address
 DX = Program name offset address

Output Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)
 AX=1: Invalid function number
 AX=2: Path or program not found
 AX=5: Access denied
 AX=8: Insufficient memory
 AX=10: Wrong environment block
 AX=11: Incorrect format

Remarks

The directory name passed is an ASCII string which is terminated by an end character (ASCII code 0). It can contain a path designation and drive specifier. No wildcards are allowed. If no drive specifier or path designation exists, the function accesses the current drive or directory.

Only EXE or COM programs can be executed. To execute a batch file, the command processor (COMMAND.COM) must be called using the /c parameter followed by the name of the batch file.

The parameter block must have the following format:

Byte 0-1: Segment address where the overlay will be stored
 (offset address=0)

Byte 2-3: Relocation factor

The relocation factor requires the value 0 for COM programs. Use the segment address at which the program should load when accessing EXE programs. The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 4BH, Subfunction 05H	DOS (Version 5.0 and above)
Set EXECSTATE	

Applications which load other programs or overlays without the DOS Exec function, must use this function after version 5.0 of DOS to avoid problems during loading of the programs and overlays.

Input AH = 4BH
 AL = 05H
 DS:DX = FAR-pointer to the Exec-State structure

Output Carry-Flag=0: o.k.
 Carry-Flag=1: Error, in this case

AX = Error-Code

- 1: unknown function code
- 2: program not found
- 3: program not found
- 4: too many files opened
- 5: access denied
- 8: insufficient storage area
- 10: wrong Environment-Block
- 11: wrong format

Remarks

The call of this function must occur between the loading of the program or overlay and its execution. Between the call of this function and the start of the program or overlay, neither DOS- nor BIOS functions, nor any other software interrupts may be called.

In the ExecState structure, information about the overlay or program is stored. It consists of 18 Bytes and must have the following format.

Format of the ExecState structure		
Addr.	Content	Type
+00H	Reserved, must contain 0	1 WORD
+02H	1 = EXE-program 2 = Overlay	1 WORD
+04H	Pointer to an ASCIIZ string with the name of the program or overlay (Path not allowed in this string)	1 PTR
+08H	Segment address of the PSP of the of the program or overlay	1 WORD
+0AH	Jump location to the program or overlay	1 PTR
+0EH	Program- or overlay size including PSP	1 DWORD
Length: 12H (18 Bytes)		

Interrupt 21H, Function 4CH	DOS (Version 2.0 and above)
Terminate with return code	

Terminates a program and passes an end code for which function 77 (4DH-see below) searches. This function releases the memory previously occupied by the terminated program.

Input AH = 4CH

AL = Return code

Output No output

Remarks

This function may be used for program termination instead of the other functions listed earlier.

This function call restores the contents of the three interrupt vectors that were stored in the PSP when the program started execution.

Before passing control to the calling program, all handles opened by this program close, along with the corresponding files. This is not applicable to files accessed using FCBs.

A batch file can test for the return code using the ERRORLEVEL and IF batch commands.

Interrupt 21H, Function 4DH	DOS (Version 2.0 and above)
Get return code	

Checks a program, called from another program by the EXEC function, for the return code passed by the called program when it terminates.

Input	AH = 4DH
Output	AH = Type of program termination
	AH=0: Normal end
	AH=1: End through Ctrl C or Break
	AH=2: Device access error
	AH=3: Call of function 49 (31H)
	AL = Return code

Remarks

This function reads the return code of the called program only once.

The contents of the AX, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and flag registers are not affected by this function. The contents of all other registers may change.

Interrupt 21H, Function 4EH	DOS (Version 2.0 and above)
Search for first match	

Searches for the first occurrence of the filename listed. The file can have certain attributes, so a search can be made through subdirectories and volume names.

Input	AH = 4EH
CX =	File attribute
DS =	Filename segment address
DX =	Filename offset address

Output Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=2: Path not found

AX=18: No file with the attribute found

Remarks

The directory name passed is an ASCII string which is terminated by an end character (ASCII code 0). It can contain a path designation and drive specifier. No wildcards are allowed. If no drive specifier or path designation exists, the function accesses the current drive or directory.

The search defaults to normal files (attribute 0). Any set attribute bits extends the search to normal files and any other file types.

If a matching file occurs, the first 43 bytes of the DTA contain the following information about this file:

Bytes 0-20: Reserved
 Byte 21: File attribute
 Bytes 22-23: Time of last modification to file
 Bytes 24-25: Date of last modification to file
 Bytes 26-27: Low word of file size
 Bytes 28-29: High word of file size
 Bytes 30-42: ASCII filename and extension terminated
 by an end character (ASCII code 0)

This function may only be called to search for the first occurrence of a file. If you want to search for a group of files using wildcards, function 4FH (see below) must be called.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 4FH

DOS (Version 2.0 and above)

Search for next match (handle)

Searches for subsequent occurrences of the filename listed after function 78 (above) executed successfully.

Input AH = 4FH

Output Carry flag=0: O.K.

Carry flag=1: Error (AX=Error code)

AX=18: No other files found with this attribute

Remarks

If a matching file occurs, the first 43 bytes of the DTA contain the following information about this file:

Bytes 0-20: Reserved

Byte 21: File attribute

Bytes 22-23: Time of last modification to file

Bytes 24-25: Date of last modification to file

Bytes 26-27: Low word of file size

Bytes 28-29: High word of file size

Bytes 30-42: ASCII filename and extension terminated by an end character (ASCII code 0)

This function can only be called if function 4EH has been called once and if the DTA remains unchanged.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21(h), function 50(h) DOS

Reserved (Version 2 and up)

Interrupt 21(h), function 51(h) DOS

Reserved (Version 2 and up)

Interrupt 21(h), function 52(h) DOS

Reserved (Version 2 and up)

Interrupt 21(h), function 53(h) DOS

Reserved (Version 2 and up)

Interrupt 21H, Function 50H

DOS (Version 2.0 and above)

Set active PSP

This function sets the current PSP, as opposed to the DOS. It is required for example in TSR programs to obtain data during file access from you PSP and not from the PSP of the current foreground program.

Input

AH = 50H

BX = Segment address of the PSP

Output

Carry-Flag=0: o.k.

Carry-Flag=1: Error, in this case

AX = Error-code

Remarks

DOS assumes that a PSP always starts at the Offset address 0000H in a segment. This should be noted during the call of this function.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 51H**DOS (Version 2.0 and above)**

Determine active PSP

With this function the Segment address of the PSP of the current program can be determined. This is mostly used by TSR programs which note the address of the PSP of the foreground program, before setting their own PSP with the help of the function 50H.

Input AH = 51H

Output Carry-Flag=0: o.k. in this case

BX = Segment address of the current PSP

Carry-Flag=1: Error, in this case

AX = Error-code

Remarks

The indicated PSP always begins at the Offset address 0000H relative to the indicated Segment address.

Function 62H should be preferred to the call of this function, which is specifically designed for this purpose and in contrast to function 51H is documented openly. However, it is only available after version 3.0 of DOS, while the function 51H was already introduced with version 2.0.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 53H**DOS (Version 2.0 and above)**

Convert BPB to DPB

This function returns a pointer to the DOS-Info-Block (DIB). With this function much interesting information can be obtained, which otherwise would not be accessible to a program.

Input AH = 52H

Output Carry-Flag=0: o.k. in this case

ES:BX =FAR-pointer to the DIB

Carry-Flag=1: Error, in this case

AX = Error-code

Remarks

The structure of the DOS-Info-Block can be found in Chapter 6.

The contents of the CX, DX, SI, DI, BP, CS, DS and SS registers are not affected by this function.

Interrupt 21H, Function 53H**DOS (Version 2.0 and above)**

Convert BPB to DPB

This undocumented function converts an available BIOS-Parameter-Block (BPB) to a Drive-Parameter-Block (DPB).

Input	AH = 53H
	DS:SI = FAR-pointer to the buffer with the BPB to be converted
	ES:BP = Pointer to the buffer, in which the Drive-Parameter-Block should be created (see below)
Output	Carry-Flag=0: o.k.
	Carry-Flag=1: Error, in this case

AX = Error-code

Remarks

A description of the structure of the BIOS and the Drive-Parameter-Block, can be found in Chapter 6.

Interrupt 21H, Function 54H	DOS (Version 2.0 and above)
Get verify flag	

Gets the current status of the verify flag. This flag determines whether or not data transmitted to a medium (floppy disk or hard drive) should be verified after the transmission.

Input	AH = 54H
Output	AL = Verify flag
	AL=0: Verify off
	AL=1: Verify on

Remarks

Function 2EH (see above) controls the status of the verify flag.

The contents of the AH, BX, CX, DX, SI, DI, BP, CS, DS, SS, ES and flag registers are not affected by this function.

Interrupt 21H, Function 55H	DOS (Version 2.0 and above)
Create new PSP	

This function creates a new PSP by copying the indicated old PSP into the new one and then adding various new information into the new PSP. This new information is mainly the information which is dependent on the position of the PSP in storage and therefore can not be accepted unchanged from the old PSP into the new one.

Input	AH = 55H
	DX = Segment address of the new PSP
	CX = Segment address of the old PSP
Output	Carry-Flag=0: o.k.
	Carry-Flag=1: Error

Remarks

This undocumented function was created to be able to create a PSP for a program before loading the program. In view of the function 4BH, it is not required any more, because function 4BH loads the program and also automatically creates a new PSP that includes the adjustment of the Segment address in the PSP and in the program that was loaded.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 56H**DOS (Version 2.0 and above)**

Rename file (handle)

Renames a file or moves the file to another directory of a block device. Moving is possible only within the different directories of one particular device (i.e., you can't move a file from a hard drive directory to a floppy disk directory).

Input

AH = 56H

DS = Old filename segment address

DX = Old filename offset address

ES = New filename segment address

DI = New filename offset address

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=2: File not found

AX=3: Path not found

AX=5: Access denied

AX=11: Not the same device

Remarks

The directory name passed is an ASCII string which is terminated by an end character (ASCII code 0). It can contain a path designation and drive specifier. No wildcards are allowed. If no drive specifier or path designation exists, the function accesses the current drive or directory.

An error occurs if you attempt to move the file to a filled root directory.

This function cannot access subdirectories or volume names.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 57H, Subfunction 0**DOS (Version 2.0 and above)**

Get file date and time

Gets the date and time of the creation or last modification of a file.

Input

AH = 57H

AL = 0

BX = Handle

Output Carry flag=0: O.K.

CX=Time

DX=Date

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function

AX=6: Invalid handle

Remarks

For it to be accessed with a handle, the file must have been previously opened or created using one of the handle functions.

The time appears in the CX register in the following format:

Bits 0-4: Seconds in 2-second increments

Bits 5-10: Minutes

Bits 11-15: Hours

The date appears in the DX register in the following format:

Bits 0-4: Day of the month

Bits 5-8: Month

Bit 9-15: Year (relative to 1980)

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 57H, Subfunction 1

DOS (Version 2.0 and above)

Set file date and time

Sets the date and time of the creation or last modification of a file in the corresponding file and device.

Input AH = 57H

AL = 1

BX = Handle

CX = Time

DX = Date

Output Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function

AX=6: Invalid handle

Remarks

To be accessed with a handle, the file must have been previously opened or created using one of the handle functions.

The time appears in the CX register in the following format:

Bits 0-4: Seconds in 2-second increments

Bits 5-10: Minutes

Bits 11-15: Hours

The date appears in the DX register in the following format:

Bits 0-4: Day of the month

Bits 5-8: Month

Bit 9-15: Year (relative to 1980)

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 58H, Subfunction 0**DOS (Version 3.0 and above)**

Get allocation strategy

Determines the method currently in use by MS-DOS for allocating blocks of memory. If a program allocates memory using function 48H, different programs in memory may already have memory blocks assigned to them. Since these requested memory blocks vary in size, DOS has three methods of allocating memory to a program:

First fit: DOS starts searching at the start of memory and allocates the first memory block it finds of the requested size.

Best fit: DOS searches all available memory blocks and allocates the smallest suitable memory block it finds (the most efficient method).

Last fit: DOS starts searching at the end of memory and allocates the first memory block it finds of the requested size.

Input

AH = 58H

AL = 0

Output

Carry flag=0: O.K.

AX=0: First fit (start from beginning of memory)

AX=1: Best fit (search for best-fitting memory block)

AX=2: Last fit (start from end of memory)

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function number

Remarks

The allocation strategy applies to all programs.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 58H, Subfunction 1	DOS (Version 3.0 and above)
Get allocation strategy	

Defines the method used by MS-DOS for allocating blocks of memory. If a program allocates memory using function 48H, different programs in memory may already have memory blocks assigned to them. Since these requested memory blocks vary in size, DOS has three methods of allocating memory to a program:

First fit: DOS starts searching at the start of memory and allocates the first memory block it finds of the requested size.

Best fit: DOS searches all available memory blocks and allocates the smallest suitable memory block it finds (the most efficient method).

Last fit: DOS starts searching at the end of memory and allocates the first memory block it finds of the requested size.

Input

AH = 58H

AL = 1

BX = Allocation strategy

BX=0: First fit (start from beginning of memory)

BX=1: Best fit (search for best-fitting memory block)

BX=2: Last fit (start from end of memory)

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function number

Remarks

The allocation strategy applies to all programs.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 58H, Subfunction 02H	DOS (Version 5.0 and above)
Query the inclusion of the UMBs	

Starting with the version 5.0, the Upper-Memory-Blocks (UMBs), which are located between 640 KB and the 1-MB border, can be included in the DOS storage administration. This function informs the caller, if the UMBs are participating at this time in the storage administration.

Input

AH = 58H

AL = 02H

Output

Carry-Flag=0: o.k., in this case

AL = 0 : UMBs are not used

AL = 1 : UMBs are included in the storage administration

Carry-Flag=1: Error, in this case

AX = 1: no UMB support, because the MS-DOS=UMB command was not indicated

AX = 7: storage administration destroyed

Remarks

The error-code 7 is returned when DOS notices an inconsistency in its storage administration, which is usually caused by an erroneous access in a DOS program to this storage area.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 58H, Subfunction 03H	DOS (Version 5.0 and above)
Determine inclusion of the UMBs	

Starting with the Version 5.0, the Upper-Memory-Blocks (UMBs), which are located between 640 KB and the 1-MB border, can be included in the DOS storage administration. This function determines if the UMBs are participating at this time in the storage administration.

Input

AH = 58H

AL = 03H

BX = 0 : UMBs not participating

1 : UMBs are participating in the storage administration

Output

Carry-Flag=0: o.k.

Carry-Flag=1: Error, in this case

AX = 1: no UMB support, because the MS-DOS=UMB command was not indicated

AX = 7: storage administration destroyed

Remarks

The error-code 7 is returned when DOS notices an inconsistency in its storage administration, which is usually caused by an erroneous access in a DOS program to this storage area.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 59H	DOS (Version 3.0 and above)
Get extended error information	

Gets information about errors that occur during the call of one of the functions of either interrupt 21H or interrupt 24H. This information includes detailed information about the error, its origin and the action the user should take to alleviate the error.

Input

AH = 59H

BX = 0

Output

AX = Description of error

BH = Cause of error

BL = Recommended action

CH = Source of error

Remarks

The following codes describe the error:

Code	Error	Code	Error
0:	No error	20:	Unknown device
1:	Invalid function number	21:	Device not ready
2:	File not found	22:	Unknown command
3:	Path not found	23:	CRC error
4:	Too many files open at once	24:	Bad request structure length
5:	Access denied	25:	Seek error
6:	Invalid handle	26:	Unknown medium type
7:	Memory control block destroyed	27:	Sector not found
8:	Insufficient memory	28:	Printer out of paper
9:	Invalid memory address	29:	Write error
10:	Invalid environment	30:	Read error
11:	Invalid format	31:	General failure
12:	Invalid access code	32:	Sharing violation
13:	Invalid data	33:	Lock violation
14:	Reserved	34:	Unauthorized disk change
15:	Invalid drive	35:	FCB not available
16:	Current directory cannot be removed	80:	File already exists
17:	Different device	81:	Reserved
18:	No additional files	82:	Directory cannot be created
19:	Medium write protected	83:	Terminate after call of interrupt 24H

The following codes describe the cause of the error:

Code	Error	Code	Error
1:	No memory available on the medium	7:	Application program error
2:	Temporary access problem-may end soon	8:	File not found
3:	Access unauthorized	9:	Invalid file format/type
4:	Internal error in system software	10:	File locked
5:	Hardware error	11:	Wrong medium in drive, bad disk or medium problem
6:	Software failure not caused by running application program	12:	Other error

The following codes describe the source of the error:

Code	Error	Code	Error
1:	Unknown	4:	Serial device
2:	Block device (disk drive, hard drive, etc.)	5:	RAM
3:	Network		

The contents of the CS, DS, SS and ES registers are not affected by this function. All other register contents are destroyed.

Interrupt 21H, Function 5AH

DOS (Version 3.0 and above)

Create temporary file (handle)

Creates a temporary file in memory for storage during program execution. The filename doesn't matter because the access occurs through the assigned handle. Since this function allows several files open at the same time, DOS creates filenames from the current date and time. Every temporary file is ensured its own particular name because the function cannot be called more than once at a time.

Input

AH = 5AH
 CX = File attribute
 DS = Directory segment address
 DX = Directory offset address

Output

Carry flag=0: O.K.

AX=Handle

DS=Complete filename segment address

DX=Complete filename offset address

Carry flag=1: Error (AX = Error code)

AX=3: Path not found

AX=5: Access denied

Remarks

The directory name passed is an ASCII string which is terminated by an end character (ASCII code 0). It can contain a path designation and drive specifier. No wildcards are allowed. If no drive specifier or path designation exists, the function accesses the current drive or directory.

The bits of the file attribute have the following meanings:

Bit 0 = 1: Read only file

Bit 1 = 1: Hidden file

Bit 2 = 1: System file

Temporary files are not automatically deleted after program execution. The file must be closed using function 3EH, then the temporary file must be deleted using function 41H.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function.

Interrupt 21H, Function 5BH**DOS (Version 3.0 and above)**

Create new file (handle)

Creates a file in the specified directory based upon an ASCII file format. If no drive specifier or path is provided, the file opens in the default (current) directory.

Input

AH = 5BH

CX = File attributes:

CX=00: Normal file

CX=01: Read-only file

CX=02: Hidden file

CX=04: System file

DS = ASCII file specification segment address

DX = ASCII file specification offset address

Output

Carry flag=0 (AX= file handle)

Carry flag=1 (AX = Error code)

AX=3: Path not found

AX=4: No handle available

AX=5: Access denied

AX=80 (50H): File already exists

Remarks

An error occurs when any element of the path designation doesn't exist, when the filename already exists in the specified directory, or when an attempt is made to create the file in an already full root directory.

The file defaults to the normal read/write attribute, which allows both read and write operations. This attribute can be changed by using function 43H.

Interrupt 21H, Function 5CH**DOS (Version 3.0 and above)**

Control record access

Locks or unlocks a particular section of a file. This function operates on multitasking and networking systems.

Input

AH = 5CH

AL = Function code

AL=00: Lock file section

AL=01: Unlock file section

BX = File handle
 CX = High word of section offset
 DX = Low word of section offset
 SI = High word of section length
 DI = Low word of section length

Output

Carry flag=0: Successful lock/unlock
 Carry flag=1: Error (AX = Error code)
 AX=1: Invalid function code
 AX=6: Invalid handle
 AX=33 (21H): All or part of section already locked

Remarks

This function can only be used on files already opened or created using functions 3CH, 3DH, 5AH or 5BH.

The corresponding call to unlock a file region must contain the identical file offset and file region length.

Interrupt 21H, Function 5CH, Subfunction 01H**DOS (Version 3.0 and above)**

Release of a blocked area in a file

With this function, areas in a file which had been blocked previously with the help of subfunction 00H, can be released again.

Input

AH = 5CH
 AL = 01H
 BX = Handle of the file
 CX = Hi-word of the address of the first byte in the file, which should be released again
 DX = Lo-Word, of the address of the first byte to be released
 SI = Hi-Word of the number of bytes to be released
 DI = Lo-Word of the number of bytes to be released
 Carry-Flag=0: o.k.
 Carry-Flag=1: Error, in this case AX = Error-Code
 1: Network software is not active
 6: invalid Handle
 33: the area indicated is not blocked

Remarks

This function can only be called after the SHARE-command or other network-software was previously called.

The Start-offset of the region to be blocked and its length are indicated as positive LONGINTS, which consist of 2 words and therefore 4 bytes. They must agree exactly with the indications that were made during a preceding call of the subfunction 00H.

The contents of the BX, CX, DX, SI, DI, BP and the segment registers are not affected by this function.

Interrupt 21H, Function 5DH	DOS (Version 1.0 and above)
Reserved	

Interrupt 21H, Function 5EH, Subfunction 0	DOS (Version 3.1 and above)
Get machine name	

Returns the address of an ASCII string which defines the local computer type within a network.

Input

AH = 5EH

AL = 00

DS = User buffer segment address

DX = User buffer offset address

Output

Carry flag=0: Successful execution

CH = 00: Name undefined

CH > 00: Name defined

CL = NETBIOS name number (when CH<>00)

DS = Identifier segment address (when CH<>00)

DX = Identifier offset address (when CH<>00)

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function code

Remarks

The computer type is a 15-byte-long string terminated by an end character (ASCII code 0).

Interrupt 21H, Function 5EH, Subfunction 2	DOS (Version 3.1 and above)
Set printer setup	

Specifies a string which precedes all output to a particular printer used by a network. This string allows network users to assign their own individual printing parameters to the shared printer.

Input

AH = 5EH

AL = 02

BX = Redirection list index (see Remarks below)

CX = Printer setup string length

DS = Printer setup string segment address

SI = Printer setup string offset address

Output Carry flag=0: Successful execution

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function code

Remarks

The contents of register BX (redirection list index) come from function 94 5EH, subfunction 2. Function 5EH, subfunction 3 (see below) can supply the current printer setup string.

Interrupt 21H, Function 5EH, Subfunction 3

DOS (Version 3.1 and above)

Get printer setup

Gets the printer setup string assigned to a particular network printer by using function 5EH, subfunction 2 (see above).

Input

AH = 5EH

AL = 03

BX = Redirection list index)

DS = Setup string receiving buffer segment address

SI = Setup string receiving buffer offset address

Output Carry flag=0: Successful execution

CX=Printer setup string length

ES=Segment address of buffer retaining setup string

DI=Offset address of buffer retaining setup string

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function code

Remarks

The contents of register BX (redirection list index) come from function 5EH, subfunction 2. Function 5EH, subfunction 3 can supply the current printer setup string.

Interrupt 21H, Function 5FH, Subfunction 2

DOS (Version 3.1 and above)

Get redirection list entry

Gets the system redirection list. This list assigns local names to network printers, files or directories.

Input

AH = 5FH

AL = 02

BX = Redirection list index (see Remarks below)
 DS = Device name buffer segment address (16 bytes)
 SI = Device name buffer offset address (16 bytes)
 ES = Network name buffer segment address (128 bytes)
 DI = Network name buffer offset address (128 bytes)

Output

Carry flag=0: Successful execution

BH = Status flag

BH=0: Valid device

BH=1: Invalid device

BL = Device type

BL=3: Printer

BL=4: Drive

BP = Destroyed

CX = Parameter value in memory

DX = Destroyed

DS = ASCII format local device name segment address

SI = ASCII format local device name offset address

ES = ASCII format network name segment address

DI = ASCII format network name offset address

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function code

AX=18: No more files available

Remarks

The contents of register CX come from function 5FH, subfunction 3 (see below).

Interrupt 21H, Function 5FH, Subfunction 3**DOS (Version 3.0 and above)**

Redirect device

Redirects device access in a network, assigning a network name to a local device.

Input

AH = 5FH

AL = 03

BL = Device type

BL=3: Printer

BL=4: Drive

CX = Parameter value in memory

DS = ASCII format local device name segment address

SI = ASCII format local device name offset address

ES = ASCII format network name and password segment address

DI = ASCII format network name and password offset address

Output

Carry flag=0: Successful execution

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function code; string format incorrect; device redirected

AX=3: Path not found

AX=5: Access denied

AX=8: Insufficient memory

Remarks

The contents of register CX are taken from function 5FH, subfunction 3.

Device names can be drive specifiers (e.g., A:), printer names (i.e., LPT1, PRN, LPT2 or LPT3) or null strings. If you enter a null string and password as the device name, DOS tries to open access to the network using the password.

Interrupt 21H, Function 5FH, Subfunction 4**DOS (Version 3.0 and above)**

Cancel redirection

Disables the current redirection by removing local name assignments to network printers, files or directories.

Input

AH = 5FH

AL = 04

BX = Redirection list index (see Remarks below)

DS = ASCII format local device name segment address

SI = ASCII format local device name offset address

Output

Carry flag=0: Successful execution

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function code; device name not on network

AX=15: Redirection halted

Remarks

Device names can be drive specifiers (e.g., A:), printer names (i.e., LPT1, PRN, LPT2 or LPT3) or strings beginning with double backslashes (i.e., \\). A string preceded by two backslashes terminates communications between the local computer and the network.

Interrupt 21H, Function 62H**DOS (Version 3.0 and above)**

Get PSP address

Gets the segment address of the PSP from the currently executing program.

Input

AH = 62H

BX = PSP segment address

Remarks

The PSP starts at address BX:0000.

The contents of the AX, CX, DX, SI, DI, BP, CS, DS, SS, ES registers and the flag registers are not affected by this function.

Interrupt 21H, Function 63H, Subfunction 0**DOS (Version 2.25 only)**

Get lead byte table

Gets the address of the system table which defines the byte ranges for the PC's extended character sets.

Input

AH = 9963H

AL = 00: Get address of system lead byte table

Output

DS = Table segment address

SI = Table offset address

Remarks

This function is available only in DOS Version 2.25.

Interrupt 21H, Function 63H, Subfunction 1**DOS (Version 2.25 only)**

Set or clear interim console flag

Clears the interim console flag.

Input

AH = 63H

AL = 01: Clear or set interim console flag

DL = Interim console flag setting

DL=01: Set interim console flag

DL=00: Clear interim console flag

Output

No output

Remarks

This function is available only in DOS Version 2.25.

Interrupt 21H, Function 63H, Subfunction 2	DOS (Version 2.25 only)
Gets the interim console flag	

Gets the interim console flag.

Input	AH = 63H
	AL = 02: Get interim console flag value
Output	DL = Flag value

Remarks

This function is available only in DOS Version 2.25.

Interrupt 21H, Function 64H	DOS (Version 3.0 and above)
Reserved	

Interrupt 21H, Function 65H	DOS (Version 3.3 and above)
Get extended country information	

Gets information about the specific country/code page.

Input	AH = 65H
	AL = subfunction:
	AL = 1: Get international information
	AL = 2: Get uppercase pointer table
	AL = 4: Get pointer to uppercase pointer table (filename)
	AL = 6: Get pointer to collation table
	BX = Code page:
	BX = -1: active CON device
	CX = Length of buffer allocated to receive information
	DX = Country ID number
	DX = -1: Default
	ES:DI = Address of buffer allocated to receive information
Output	Carry flag=0: Successful execution
	Carry flag=1: Error (AX = Error code)

Remarks

The information this function returns is an extended version of the information returned by int 21H, function 38H.

An error may occur if the country code in DX is invalid, or if the code page number is different from the country code, or if the buffer length specified in the CX register is less than five bytes. If the buffer is not long enough to receive all the information, the function accepts as much information as the buffer will accept. This buffer contains the following information after the call:

Byte 0: ID code for information

Bytes 1-2: Length of buffer

Bytes 3-4: Country ID

Bytes 5-6: Code page

Bytes 7-8: Date format

0 = USA: Month-day-year

1 = Europe: Day-month-year

2 = Japan: Year-month-day

Bytes 9-13: Currency indicator

Bytes 14-15: ASCII code of the thousand character (comma/period)

Bytes 16-17: ASCII code of the decimal character (period/comma)

Bytes 18-19: ASCII code of the date separation character

Bytes 20-21: ASCII code of the time separation character

Byte 22: Currency format

bit 0 = 0: Currency symbol before the value

bit 0 = 1: Currency symbol after the value

bit 1 = 0: No spaces between value and currency symbol

bit 1 = 1: Space between value and currency symbol

Byte 23: Precision (number of decimal places)

Byte 24: Time format

bit 0 = 0: 12-hour clock

bit 0 = 1: 24-hour clock

Bytes 25-28: Address of character conversion routine

Bytes 29-30: ASCII data separator

Bytes 31-40: Reserved

Interrupt 21H, Function 66H**DOS** (*Version 3.3 and above*)

Get or set code page

Gets or sets the current code page.

Input

AH = 66H

AL = subfunction:

AL = 1: Get code page

AL = 2: Select code page

BX = Selected code page (if AL = 2)

Output

Carry flag=0: Successful execution

If AL = 1 used for input:

BX = active code page

DX = default code page

Carry flag=1: Error (AX = Error code)

Remarks

If subfunction 2 is used, COUNTRY.SYS supplies the code page number.

The DEVICE... (CONFIG.SYS), NLSFUNC and MODE CP PREPARE commands (AUTOEXEC.BAT) must have already configured the system for code page switching before this function may be called.

Interrupt 21H, Function 67H**DOS** (*Version 3.3 and above*)

Set handle count

Sets the maximum number of accessible files and devices that may be currently opened using handles.

Input

AH = 67H

BX = Number of handles desired

Output

Carry flag=0: Successful execution

Carry flag=1: Error (AX = Error code)

Remarks

The PSP's default table reserved for the process can control 20 handles.

An error occurs if the content of the BX register is greater than 20, or if insufficient memory exists to allocate a block for the extended table.

If the number in the BX register is greater than the number of entries assigned by the FILES entry in the CONFIG.SYS file, no error occurs. However, attempts at opening a file or device fail if all file entries are in use, even if file handles are still available.

Interrupt 21H, Function 68H	DOS (Version 3.3 and above)
Commit file	

Writes all DOS buffers associated to a specific handle to the specified device. If the handle points to a file, the file's contents, date and size are updated.

Input AH = 68H
 BX = File handle

Output Carry flag=0: Successful execution
 Carry flag=1: Error (AX = Error code)

Remarks

This function performs the same task as closing and reopening a file or duplicate handle, even without handles. If this function accesses a character device's handle, the carry flag returns 0 but nothing else happens.

Multiprocessing and networking applications maintain control of the file.

Interrupt 21H, Function 69H	DOS (Version 1.0 and above)
Reserved	

Interrupt 21H, Function 6AH	DOS (Version 1.0 and above)
Reserved	

Interrupt 21H, Function 6BH	DOS (Version 1.0 and above)
Reserved	

Interrupt 21H, Function 6CH	DOS (Version 4.0 and above)
Extended Open function	

Starting with Version 4.0 of DOS there is the capability to create a file during the OPEN-call, or to overwrite an already existing file. The function therefore enhances the capability of the normal OPEN function 3DH.

Input AH = 3DH
 AL = 0
 BX = access mode
 CX = file attribute
 DX = DOS reaction
 DS:SI = FAR-pointer to the buffer with the filename

Output Carry-Flag=0: o.k., in this case AX = Handle of the file
 CX = Status

Carry-Flag=1: Error, in this case AX = Error-Code

- 1: File-Sharing software missing
- 2: file not found
- 3: Path not found or file does not exist
- 4: no more free Handles
- 5: access denied
- 12: this access mode not allowed

Remarks

The filename must be available as an ASCII string which is terminated with an end character (ASCII-Code 0). Besides a device designation it may contain a complete Path designation and a filename, but not a Wildcard. If the device designation or Path description is missing, access occurs to the current device or the current Directory.

The access mode is constructed as follows:

Bit 0 - 2: Read/Write permission

- 000b = file can only be read
- 001b = file can only be written
- 010b = file can be read and written

Bit 3: 0b

Bit 4 - 6: File-Sharing mode

- 000b = only the current program may access the file (Compatibility mode)
- 001b = only the current program may access the file
- 010b = another program may read, but not write the file
- 011b = another program may write, but not read the file
- 100b = another program may read and write the file

Bit 7: Handle-Flag

- 0 = Also the child-program of the current program is allowed to access the Handle of this file.
- 1 = Only the current program can access the Handle of this file.

Bit 8 - 12: 0b

Bit 13: 0 = On a critical error call Interrupt 24h.

- 1 = On a critical error return the corresponding error code to AX, but do not call Interrupt 24h.

Bit 14 1 = For every write access to this file immediately change the Directory entry.

Bit 15: 0b

The file attribute in the CX-Register can be loaded before the function call with the following attributes:

Bit 0 = 1: file can only be read, but not written (Read-Only)

Bit 1 = 1: file is hidden (not displayed during DIR command)

Bit 2 = 1: file is a system-file

Bit 5 = 1: file has been changed since the last archiving

Bit 6-15 : 0b

The reaction of the function is stored in the lower two nibbles of the DX-Register, if the file to be opened does not exist or does exist. Bits 0 to 3 represent the reaction if the file does not yet exist. The values mean:

0000b = Terminate function with an error-code

0001b = Create file

In bits 4 to 7 the reaction to an already existing file is recorded. The values have the following meaning:

0000b = Terminate function with an error-code

0001b = Open existing file

0010b = Overwrite existing file and open

The function 3DH can continued to be used to open a file also under DOS 4.0.

If the function call was terminated in an orderly manner, all other Handle functions can be called through the passed Handle. In the CX register remains the function status which provides information about the action of the function as follows:

0 = File opened

1 = New file created and opened

2 = Existing file overwritten and opened

The file pointer is set to the first byte of the file.

The File-Sharing mode in bits 4 to 6 of the access mode is also under DOS version 4 only interesting if the file is on a mass storage device which is part of a network. In this case, these 3 bits decide, if other programs in the network may access this file during the opening and the access rights of this program. If the value 0 is indicated for these bits, DOS treats this file in the Compatibility-Mode, in which the existence of a network in relation to this file is ignored and only the current program may access this file.

Error 12 can only occur under DOS version 3 and only in a network, if the file was already opened by another program and it was determined at that time that at this moment no other program may access it.

The contents of the BX, DX, SI, DI, BP and segment registers are not affected by this function.

Interrupt 22H

DOS (Version 1.0 and above)

Terminate address

Contains the address of a routine which terminates a program. Control returns to the program that called for termination. You should never call this routine directly.

DOS stores the contents of this interrupt vector in the PSP of the program to be executed before passing control to the program. This prevents program changes to the vector, which could prevent DOS from calling the termination routine.

Interrupt 23H**DOS (Version 1.0 and above)****Ctrl Break** handler address

Contains the address of a routine which executes when the user presses **Ctrl C** or **Ctrl Break**. You should never directly call this routine.

DOS stores the contents of this interrupt vector in the PSP of the program to be executed before passing control to the program. This prevents program changes to the vector, which could prevent DOS from calling the termination routine.

Interrupt 24H**DOS (Version 1.0 and above)**

Critical error handler address

Represents a routine called during hardware access (e.g., disk drive) when a critical error occurs. You should never directly call this routine.

When an application routine is called during a critical error, bit 7 of the AH register indicates the type of failure (0 = disk/hard drive error, 1 = other errors). A disk/hard disk error will only be reported after several attempted accesses. During the call, the DI register receives one of the following codes:

0:	Disk write protected	7:	Unknown device type
1:	Access on unknown device	8:	Sector not found
2:	Drive not ready	9:	Printer out of paper
3:	Invalid command	10:	Write error
4:	CRC error	11:	Read error
5:	Bad request structure length	12:	General failure
6:	Seek error		

The error routine restores the SS, SP, DS, ES, BX, CX and DX registers to the same values that they contained during the call. During execution it can only access functions 1 to 0CH of interrupt 21H. It should be terminated by an IRET instruction and pass one of the following codes to the AL register:

0:	Ignore error	2:	Terminate program using interrupt 23H
1:	Repeat the operation	3:	Fail system call (Version 3 and up only)

If a program changes the content of this interrupt vector, the program can terminate without restoring the memory contents. Since RAM can be released and used by other programs, the critical error routine can be overwritten by another program in memory. When this occurs, a critical error could cause a system crash because a completely different code now exists at the location of the old error handler routine.

Before passing control to the program, DOS stores the contents of this interrupt vector in the PSP of the program to be executed. This prevents program changes to the vector, which could prevent DOS from calling the termination routine. During program termination, the contents of the interrupt vector pass from the PSP to the vector; then the system calls the routine.

Interrupt 25H	DOS (Version 1.0 and above)
Absolute disk read	

Reads one or more consecutive sectors from a disk or hard drive.

Input

AL = Drive specifier
 CX = Number of sectors to read
 DX = First sector to read
 DS = Buffer segment address
 BX = Buffer offset address

Output

Carry flag=0: O.K.
 Carry flag=1: Error (AX = Error code)
 AX=1: Bad command
 AX=2: Bad address
 AX=4: Sector not found
 AX=8: DMA error
 AX=16: CRC error
 AX=32: Disk controller error
 AX=64: Seek error
 AX=128: Device does not respond

Remarks

In the AL register 0 represents drive A:, 1 represents drive B:, etc.

All the sectors of the medium can be accessed. DOS itself uses this interrupt to read the root directory and the FAT of a medium. The data are read from the medium into the buffer of the calling program. After the function call, the contents of all registers, except the segment register, may change.

After the interrupt call, the stack pointer changes position because two bytes stored on the stack during the call are removed and not returned. These bytes represent the flag register, which can be read from the stack using the POPF instruction. The old value of the stack pointer can be set by adding 2 to its contents. If you omit the stack pointer correction, the stack could overflow. Because of this, you cannot call this interrupt from higher level languages. You must call it from assembly language.

The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function. The contents of all other registers may change.

Interrupt 26H	DOS (Version 1.0 and above)
Absolute disk write	

Writes one or more consecutive sectors to a disk or hard drive.

Input

AL = Device designation

CX = Number of sectors to be written

DX = First sector to be written

DS = Buffer segment address

BX = Buffer offset address

Output

Carry flag=0: O.K.

Carry flag=1: Error (AX = Error code)

AX=1: Bad command

AX=2: Bad address

AX=3: Medium write protected

AX=4: Sector not found

AX=8: DMA error

AX=16: CRC error

AX=32: Disk controller error

AX=64: Seek error

AX=128: Device does not respond

Remarks

In the drive specifier 0 represents drive A:, 1 represents drive B:, etc.

All the sectors of the medium can be accessed. DOS itself uses this interrupt to write the root directory and the FAT to a medium. The data are written from the buffer of the calling program to the medium. After the function call, the contents of all registers, except the segment register, may change.

After the interrupt call, the stack pointer changes position because two bytes stored on the stack during the call are removed and not returned. These bytes represent the flag register, which can be read from the stack using the POPF instruction. The old value of the stack pointer can be set by adding 2 to its contents. If you omit the stack pointer correction, the stack could overflow. Because of this, you cannot call this interrupt from higher level languages. You must call it from assembly language. The contents of the BX, CX, DX, SI, DI, BP, CS, DS, SS and ES registers are not affected by this function. The contents of all other registers may change.

Interrupt 27H**DOS (Version 1.0 and above)**

Terminate and stay resident

Terminates the currently executing program and returns control to the program that called the current program. Unlike other functions used for program termination, the memory used by the current program keeps the program code for later recall.

Input

CS = PSP segment address

DX = Number of bytes + 1 to be reserved

Output No output

Remarks

This function is only suitable for calling COM programs.

The number of bytes to be reserved relates to the beginning of the PSP.

The value in the DX register has no effect on memory blocks reserved by function 48H of interrupt 21H.

An error occurs during the call of this interrupt if the value in the DX register ranges from FFF1H to FFFFH.

This interrupt does not close open files.

Interrupt 2FH, Subfunction 0	DOS (Version 3.0 and above)
Get print spool intall status	

Gets current installation status of the print spooler.

Input AH = 2FH

AL = 0

Output Carry flag=0: Successful execution

AL = 0: O.K. to install

AL = 1: Don't install

AL = 255: Already installed

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function

AX=2: File not found

AX=3: Path not found

AX=4: Too many files currently open

AX=5: Access denied

AX=8: Print queue full

AX=9: Print spooler busy

AX=12: Name too long

AX=15: Invalid drive

Interrupt 2FH, Subfunction 1	DOS (Version 3.0 and above)
Send file to print spooler	

Passes a file to the print spooler.

Input

AH = 2FH
AL = 1
DS = Print packet (see below) segment address
DX = Print packet (see below) offset address

Output

Carry flag=0: Successful execution
Carry flag=1: Error (AX = Error code)
AX=1: Invalid function
AX=2: File not found
AX=3: Path not found
AX=4: Too many files currently open
AX=5: Access denied
AX=8: Print queue full
AX=9: Print spooler busy
AX=12: Name too long
AX=15: Invalid drive

Remarks

The five-byte print packet contains print spooler information. The first byte indicates the DOS version (0=Versions 3.1 to 3.3); the remaining bytes indicate the segment and offset addresses of the file specification.

Interrupt 2FH, Subfunction 2**DOS (Version 3.0 and above)**

Remove file from print queue

Deletes a file from the print spooler queue.

Input

AH = 2FH
AL = 2
DS = ASCII-format file segment address
DX = ASCII-format file offset address

Output

Carry flag=0: Successful execution
Carry flag=1: Error (AX = Error code)
AX=1: Invalid function
AX=2: File not found
AX=3: Path not found

AX=4: Too many files currently open

AX=5: Access denied

AX=8: Print queue full

AX=9: Print spooler busy

AX=12: Name too long

AX=15: Invalid drive

Remarks

This subfunction allows wildcards (? and *) in file specifications, allowing you to delete more than one file at a time from the print queue.

Interrupt 2FH, Subfunction 3

DOS (Version 3.0 and above)

Cancel all files in print queue

Cancels all files waiting in the print spooler queue for printing.

Input

AH = 2FH

AL = 3

Output

Carry flag=0: Successful execution

Carry flag=1: Error (AX = Error code)

AX=1: Invalid function

AX=2: File not found

AX=3: Path not found

AX=4: Too many files currently open

AX=5: Access denied

AX=8: Print queue full

AX=9: Print spooler busy

AX=12: Name too long

AX=15: Invalid drive

Interrupt 2FH, Subfunction 4

DOS (Version 3.0 and above)

Hold print jobs for status check

Halts all print jobs while testing for spooler status.

Input

AH = 2FH

AL = 4

Output

Carry flag=0: Successful execution

Carry flag=1: Error

DX = Number of errors

DS = Print queue segment address

SI = Print queue offset address

Remarks

The print queue segment and offset addresses point to a set of 64-byte filenames in the queue. Each entry contains an ASCII file specification.

The first filename in the queue is the file currently printing in the print spooler. The last filename in the queue has a zero in the first byte of the specification.

Multiplexer-Interrupt 2FH

Because interrupt 2FH represents an interface to various resident DOS programs and allows TSR programs to make their functions available to other programs, it is known as the multiplexer.

Interrupt 2FH, Code 01H, Function 00H	MUX
PRINT: Determine installation status	

Programs can use this function to determine whether the memory resident portion of the PRINT DOS command has been loaded.

Input AH = 01H

 AL = 00H

Output AL = FFH: PRINT installed

Remarks

If PRINT has been installed, this function will return the value FFH in the AL register. Only then can the other PRINT functions be called. This function call must therefore precede all other PRINT functions.

Interrupt 2FH, Code 01H, Function 01H	MUX
PRINT: Add file to wait list	

This function allows a program to add a file to the end of the printer wait list, to be printed from the background via PRINT.

Input AH = 00H

 AL = 01H

 DS = Segment address of FAR pointer to file info data structure

 DX = Offset address of FAR pointer to file info data structure

Output Carry flag = 0 : o.k.

 Carry flag = 1: Error, in this case

 AX = error code (see remarks)

Remarks

The data structure accessed through the pointer in DS:DX comprises 5 bytes and must adhere to the following structure:

Ofs	Meaning	Type
00H	Always 0	1 Byte
01H	FAR pointer to ASCIIZ string with filename	1 FAR pointer

Wildcards are not permitted in the specified filename. However, drive and path may be designated.

If an error occurs, the one of the following error codes will be returned:

0001H	Unknown function
0002H	File not found
0003H	Path not found
0004H	Too many files opened
0005H	File access denied
0008H	Printer wait list is full
000FH	Unknown device

This function should only be called after function 00H has indicated that the resident portion of PRINT has been installed.

Interrupt 2FH, Code 01H, Function 02H
MUX

PRINT: Delete from wait list

This function can be used to remove one or more files from the printer wait list. If the file currently being printed is also specified, print output will be stopped.

Input

AH = 01H

AL = 02H

DS = Segment address of pointer to filename as ASCIIZ string

DX = Offset address of pointer to filename as ASCIIZ string

Output

Carry flag = 0 : o.k.

Carry flag = 1: Error, in this case

AX = error code (see remarks)

Remarks

Wildcards may be used in the specified filename so that several files can be removed from the wait list at one time. The only possible error code here is 0002H, "file not found", in the event that the specified file was not included in the printer wait list. This function should only be called when function 00H has indicated that the resident portion of PRINT has been installed.

Interrupt 2FH, Code 01H, Function 03H
MUX

PRINT: Delete wait list

This function deletes all files from the printer wait list, and any current print job is stopped immediately.

Input AH = 01H

AL = 03H

Output none

This function should only be called when function 00H has indicated that the resident portion of PRINT has been installed.

Interrupt 2FH, Code 01H, Function 04H	MUX
PRINT: Stop printer output and get status	

This function can be used to determine the current contents of the printer wait list and the number of previous output errors.

Input AH = 01H

AL = 04H

Output DX = Error counter

DS:SI = FAR pointer to wait list

Remarks

Upon this function call the printer output is interrupted, and resumes after function 05H is called.

The pointer returned in DS:SI points to the buffer in which DOS retains the waiting list in a series of 64 byte entries. Each of these entries contains an ASCII string with the name of the file which is to be printed. To identify the list's end, its last entry always begins with the NUL character (00).

The contents of this list may be read, but not modified.

This function should only be called when function 00H has indicated that the resident portion of PRINT has been installed.

Interrupt 2FH, Code 01H, Function 05H	MUX
PRINT: Continue print job	

After a print job has been interrupted by function 04H, this function can be used to resume the print job.

Input AH = 01H

AL = 05H

Output none

Remarks

This function call is only useful if function 04H has previously been executed.

Interrupt 2FH, Code 01H, Function 06H	MUX
PRINT: Locate printer	

If the printer wait list contains one file, this function can be used to determine the printer to which the print job will be sent.

Input AH = 01H

AL = 06H

Output Carry flag = 0 : Wait list is empty

Carry flag = 1: Wait list is not empty, in this case

AX = 0008H

DS:SI = FAR pointer to the head of the printer driver

Remarks

The values returned by this function are confusing, since the information that's needed (the pointer to the head of the printer device driver) is returned only with a set carry flag and the error code 0008 (wait list full). If the wait list is not empty, the carry flag will be cleared. In this case the pointer to the head of the printer driver will be unavailable.

Interrupt 2FH, Code 06H, Function 00H	MUX
PRINT: Get installation status	

This function allows a program to determine whether the resident portion of the DOS command ASSIGN has been loaded.

Input AH = 06H

AL = 00H

Output AL = FFH: ASSIGN installed

Interrupt 2FH, Code 10H, Function 00H	MUX
SHARE: Get installation status	

This function indicates whether the resident portion of the DOS command SHARE has been loaded.

Input AH = 10H

AL = 00H

Output AL = FFH: SHARE installed

Interrupt 2FH, Code 1AH, Function 00H	MUX
ANSI.SYS: Determine installation status	

This function indicates whether the device driver ANSI.SYS has been installed.

Input AH = 1AH

AL = 00H

Output AL = FFH: ANSI.SYS installed

Interrupt 2FH, Code 43H, Function 00H	MUX
HIMEM.SYS: Get installation status	

This function indicates whether the device driver HIMEM.SYS has been installed. This driver allows extended memory to be used in accordance with the XMS standard.

Input	AH = 43H AL = 00H
Output	AL = 80H: HIMEM.SYS installed

Remarks

Please note that this function, in contrast to most other functions that perform installation checks, returns the value 80H when successful, instead of the value FFH.

To maintain compatibility with HIMEM.SYS, this function call is also supported by other XMS memory managers.

Interrupt 2FH, Code 43H, Function 10H	MUX
HIMEM.SYS: Determine address for XMS function calls	

Unlike other memory managers, HIMEM.SYS calls XMS functions through a FAR CALL procedure, instead of an interrupt. This function is used to determine the address of that procedure.

Input	AH = 43H AL = 10H
Output	ES:BX = FAR pointer for calling XMS functions

Remarks

This function may only be called when a preceding call of function 00H has indicated that HIMEM.SYS has been installed.

Interrupt 2FH, Code 48H, Function 00H	MUX
DOSKEY: Get installation status	

This function indicates whether the DOS program DOSKEY.COM has been loaded.

Input	AH = 48H AL = 00H
Output	AL = 00H: not installed

Remarks

DOSKEY was introduced with DOS Version 5.0.

Please note that unlike other installation check functions, the only return value defined for this function is the value indicating that DOSKEY.COM has not been installed.

Interrupt 2FH, Code 48H, Function 10H	MUX
DOSKEY: Receive user input	

A program can access the DOSKEY program to prompt and receive user input.

Input

AH = 48H

AL = 10H

DS:DX = FAR pointer to data structure (see remarks)

Output none

Remarks

The data structure referenced through the pointer in DS:DX encompasses 130 bytes and receives the user input. It must be structured according to the following pattern:

Ofs	Meaning	Type
00h	Input buffer size, must be 128	1 Byte
01h	The number of character read minus 1	1 Byte
02h	The input buffer	128 Bytes

The number of bytes that have been read is entered into the data structure by DOSKEY. Therefore, unlike the length specification in the first byte, this field doesn't need to be initialized before DOSKEY is called.

Interrupt 2FH, Code 4A11H, Function 00H
DBLSPACE: Scan drive map

Enables a program to determine whether DoubleSpace is loaded and to obtain information about the DoubleSpace driver.

Input

AX = 4A11H

BX = 00H

Output

AX = 00H: DoubleSpace installed

BX = 444DH

CL = First drive designation used by DoubleSpace (65 = A:)

CH = Number of drive designations reserved for DoubleSpace

DX = Internal DoubleSpace version number

Remarks

If the function does not return 0 in register AX then DoubleSpace is not installed and the contents of the other return registers are undefined.

The value in BX stands for the two ASCII characters "MD", i.e., Microsoft DoubleSpace.

The two return values in CH and CL come from the FirstDrive and LastDrive settings in the DoubleSpace configuration file DBLSPACE.INI. Please note that, contrary to normal conventions, CL receives the value 65 (the ASCII code for A) for drive A:, rather than 0.

The internal version number from register DX is utilized by DBLSPACE.BIN, IO.SYS, and DBLSPACE.EXE, in order to maintain consistency. If bit 15 of this number is set, it is due to omitting the /MOVE parameter when calling the DBLSPACE.SYS driver in the CONFIG.SYS file. In this case, DoubleSpace stays in memory under 640K instead of shifting into upper memory.

Interrupt 2FH, Code 4A11H, Function 01H

DBLSPACE: Scan drive map

Determines whether a drive actually has another drive hidden behind it and whether a drive is compressed.

Input

AX = 4A11H
 BX = 01H
 DL = Drive to be scanned (A: = 0)

Output

AX = 00H: DoubleSpace installed
 BL = Bit 7 = 1 : Compressed drive
 = 0 : Uncompressed drive
 Bits 0 to 6 : Number of host drive
 BH = Number of CVF file (DBLSPACE.xxx), if drive is compressed

Remarks

If the function does not return 0 in register AX then DoubleSpace is not installed, and the contents of the other return registers are undefined.

If Bit 7 in register BL is set following the function call, then the drive is compressed. In that case a new call to this function must be made, with the value returned from register BL (bits 0 to 6) as the device code in DL, to ascertain the host drive. If the value from DL is again returned in bits 0 to 6 of BL, then the drive is a "swapped" drive. The host drive in this case is the drive from the first call (the return value in BL). In the other case, the drive designation has not been swapped and is therefore genuine. Here the second function call has returned the number of the host drive in bits 0 to 6 of register BL.

A drive designation for an uncompressed drive is genuine if bit 7 of BL is not set following the function call, and bits 0 to 6 of BL contain the same device number as was given in DL when the function was called.

Interrupt 2FH, Code 4A11H, Function 02H

DBLSPACE: Swap drive designations

This function arranges the swapping of a DoubleSpace device designation with that of its host drive.

Input

AX = 4A11H
 BX = 02H
 DL = Number of compressed drive whose drive designation is to be swapped (0 = A)

Output AX = 00H: Drive designations were successfully swapped

 101H: Invalid drive designation

 102H: Drive entered is not compressed

 103H: Drive designations already swapped

 All other values: DoubleSpace not installed

Remarks

This function is intended for internal DoubleSpace use only.

Interrupt 2FH, Code 4A11H, Function 03H

DBLSPACE: For internal use only

Interrupt 2FH, Code 4A11H, Function 04H

DBLSPACE: For internal use only

Interrupt 2FH, Code 4A11H, Function 05H

DBLSPACE: Mount compressed drive

Enables the mounting of a compressed drive into the system. The call is very complicated however, which is why Microsoft recommends a direct call to DBLSPACE.EXE using the DOS Exec function. The syntax here is as follows:

```
DBLSPACE.EXE /Mount=[CVFNumber] HostDrive: [/NEWDRIVE=NewDriveDesignation:]
```

Interrupt 2FH, Code 4A11H, Function 06H

DBLSPACE: Unmount a DoubleSpace drive

Unmounts an activated DoubleSpace drive, rendering it nonexistent from the user's perspective.

Input AX = 4A11H

 BX = 06H

 DL = Drive to be deactivated (0 = A:)

Output AX = 00H: Drive deactivated

 102H: Drive entered is not compressed and cannot be deactivated

Remarks

If the function returns none of these values in Register AX, then DoubleSpace isn't installed.

The compressed drive remains unchanged on the hard drive, yet DOS no longer recognizes its former device ID.

Interrupt 2FH, Code 4A11H, Function 07H

DBLSPACE: Establish storage space

This function returns the total number of sectors in the sector heap of a compressed drive, as well as the number still free.

Input

AX = 4A11H

BX = 07H

DL = Drive designation (0 = A:)

Output

AX = 00H: All OK

DS:SI = Refers to a Dword with the total number of sectors in the sector heap

DS:SI+4 = Refers to a Dword with the number of free sectors in the sector heap

Remarks

If the function does not return 0 in register AX then DoubleSpace is not installed and the contents of the other return registers are undefined.

Interrupt 2FH, Code 4A11H, Function 08H

DBLSPACE: Obtain information about CVF file fragmentation

Returns information regarding fragmentation of all CVF files on a particular host drive.

Input

AX = 4A11H

BX = 08H

DL = Compressed drive (0 = A:)

Output

AX = 00H: OK, in this case

BX = Maximum value for fragmentation

CX = Number of additional fragmentations permitted

AX = 102H: Drive entered is not compressed and cannot be deactivated

Remarks

If the function returns none of these values in Register AX then DoubleSpace is not installed and the contents of the other return registers are undefined.

Interrupt 2FH, Code 4A11H, Function 09H

DBLSPACE: Scan for maximum number of compressed drives

This function determines the number of DISK_UNIT structures which DOS needs for maintaining DoubleSpace drives, and which DoubleSpace creates in memory during the boot process for later definition of DoubleSpace drives.

Input

AX = 4A11H

BX = 09H

DL = Device ID of any compressed drive (0 = A:)

Output

AX = 0000H: OK, in this case

CL = Number of DISK_UNIT structures allocated by DoubleSpace during bootup.

Remarks

If the function does not return 0 in Register AX then DoubleSpace is not installed and the contents of the other return registers are undefined.

Each DISK_UNIT structure takes up 96 bytes of memory. The number returned here is determined by the MaxRemovableDrives parameter from the DoubleSpace initialization file DBLSPACE.INI.

Interrupt 2FH, Code ADH, Function 80H MUX

KEYB.COM: Return version number

This function returns the version number of the current DOS keyboard driver KEYB.COM.

Input AH = ADH

AL = 80H

BX = 0

Output BH = Main version number

BL = Sub version number

Remarks

If BH and BL are 00H after the function has been called, KEYB.COM is not installed.

Interrupt 2FH, Code ADH, Function 80H	MUX
KEYB.COM: Return version number	

This function is used to set the active code page.

Input AH = ADH

AL = 81H

BX = Code page number

Output Carry flag = 0 : o.k.

Carry flag = 1: Error, in this case

AX = 0001 "unknown code page"

Output none

Remarks

The following code pages are recognized by KEYB:

Code	Character set	Code	Character set
437	USA	863	French Canadian
850	Multi-lingual (all European countries)	865	Scandinavian
860	Portuguese		

Before calling this function, you should use function 80H to determine whether KEYB.COM is active.

You'll find more information on code pages in your DOS manual. In the framework of PC System Programming these do not come into play.

Interrupt 2FH, Code B0H, Function 00H	MUX
GRAFTABL: Get installation status	

This function indicates whether the resident portion of the DOS command GRAFTABL has been loaded.

Input AH = B0H
 AL = 00H

Output AL = FFH: GRAFTABL loaded

Interrupt 2FH, Code B7H, Function 00H	MUX
APPEND: Get installation status	

This function determines whether the resident portion of the DOS command APPEND has been loaded.

Input AH = B7H
 AL = 00H

Output AL = FFH: APPEND loaded

Interrupt 2FH, Code B7H, Function 02H	MUX
APPEND: Verify DOS 5 compatibility	

Input AH = B7H
 AL = 02H

Output AX = FFFFH : DOS 5.0 compatible

Remarks

This function may only be called when function 00H has indicated that APPEND has been loaded.

Interrupt 2FH, Code B7H, Function 04H	MUX
APPEND: Get list of APPEND directories	

This function returns the various directories that have been specified as the search path for files using APPEND.

Input AH = B7H

AL = 04H

Output ES:DI = FAR pointer to the buffer containing the APPEND directories.

Remarks

The buffer identified by the pointer in ES:DI, returned by the function, contains the various APPEND directories as ASCIIZ strings. As with the PATH command, the individual directories are separated by semicolons. The contents of this buffer must not be changed by the user.

This function may be called only when function 00H has indicated that APPEND has been loaded.

Interrupt 2FH, Code B7H, Function 06H

MUX

APPEND: Determine operation mode

This function returns the operation mode specified in the DOS command line at the APPEND call, using the different function switches.

Input AH = B7H

AL = 06H

Output BX = APPEND flag

Remarks

The individual bits of the returned flag represent the different operational modes. Bits that are not included in the following chart has no meaning and contain the value 0.

Bit	Description
0	1 = APPEND is active
12	1 = the APPEND directories will be included in the search only if a drive is specified with the file that is to be found, and if that drive specification corresponds to the drive declared in that particular APPEND string.
13	1 = the switch /PATH:ON is active
14	1 = the switch /E is active
15	1 = the switch /X:ON is active

This function may be called only when function 00H has indicated that APPEND has been loaded.

Interrupt 2FH, Code B7H, Function 07H

MUX

APPEND: Set operation mode

This function is the counterpart of function 06H and sets the different APPEND options.

Input

AH = B7H

AL = 07H

BX = APPEND flag

Output

none

Remarks

The bit values and their meaning for the append flag can be taken from the chart listed for function 06H.

This function may be called only when function 00H has indicated that APPEND has been loaded.

EMM Functions

The EMS standard which was introduced by Lotus, Intel and Microsoft. It defines the access to the memory expansion cards which operate on the "bank switching" principle which only accesses a small part of the whole memory expansion.

Most EMS cards in circulation support version 3.2 of this standard and therefore the EMS-functions, which were defined under Version 3.0 and 3.2. Only a few memory cards support the EMS-specification 4.0 and the functions connected with it. This is not true for most of the commercial EMS emulators, which simulate the EMS memory with Extended Memory. They're usually similar to version 4.0.

Interrupt 67H, Function 40H	LIM/EMS
Expanded memory: Get status	

Returns the error status of the EMM after calling any EMS functions.

Input	AH = 40H
Output	AH = EMM status
	AH=00H: O.K.
	AH=80H: Internal error, EMM possibly destroyed
	AH=81H: EMS hardware error

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information). This function should be the first EMM call a program makes, to ensure that the hardware and software are functioning properly.

Interrupt 67H, Function 41H	LIM/EMS
Expanded memory: Get segment address of the page frame	

Determines the segment address of the page frame.

Input	AH = 41H
Output	AH = 0: O.K.
	BX = Page frame segment address
	AH > 0: Error
	AH=80H: Internal error, EMM possibly destroyed
	AH=81H: EMS hardware error

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information). The addresses of the four physical pages can be calculated from this segment address, whereby the first page starts at address PAGE_FRAME:0000. The three other pages follow at 16K intervals.

Interrupt 67H, Function 42H	LIM/EMS
Expanded memory: Get number of EMS pages	

Informs the calling program how many 16K EMS pages are installed, and how many EMS pages are still available or unallocated.

Input AH = 42H

Output AH = 0: O.K.

 BX = Number of free (unallocated) pages

 DX = Total number of EMS pages

 AH > 0: Error

 AH=80H: Internal error, EMM possibly destroyed

 AH=81H: EMS hardware error

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information). The number of kilobytes of free EMS memory can be calculated by multiplying the number of free pages by 16.

Interrupt 67H, Function 43H	LIM/EMS
Expanded memory: Allocate EMS memory	

Allocates a given number of 16K EMS pages for later access.

Input AH = 43H

 BX = Number of logical (16K) pages to be allocated

Output AH = 0: O.K.

 DX = Handle for accessing allocated memory

 AH > 9: Error

 AH=80H: Internal error, EMM possibly destroyed

 AH=81H: EMS hardware error

 AH=85H: No more handles available

 AH=87H: Not enough pages free

 AH=88H: No pages were requested

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The handle returned can be used for future access and for releasing the allocated memory. If this handle is "lost", the handle cannot be recovered, nor can memory be released or used by other programs.

A call to this function may fail because there are not enough pages free or because the EMM has been called so often that no more handles are available.

The handles normally have the numbers FF00H, FE01H, FD02H, FC03H, etc.

Interrupt 67H, Function 44H	LIM/EMS
Expanded memory: Set mapping	

Places one of the pages previously allocated by function 43H in one of the four physical pages within the page frame.

Input

AH = 44H
 AL = Physical page number (0 to 3)
 BX = Logical page number
 DX = Handle

Output

AH = Error status
 AH=00H: O.K.
 AH=80H: Internal error, EMM possibly destroyed
 AH=81H: EMS hardware error
 AH=83H: Invalid handle
 AH=8AH: Invalid logical page
 AH=8BH: Invalid physical page

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The handle used when calling this function must have been returned by a previous call to EMM function 43H.

The logical pages are numbered from 0 on, so that the value 0 must be passed to access the first logical page. The largest value allowed is the number of allocated pages minus one.

Before accessing the physical page, the segment address of the page frame must be determined with function 41H.

Interrupt 67H, Function 45H	LIM/EMS
Expanded memory: Release pages	

Releases pages allocated with function 43H to the EMM. This makes these pages available to other applications.

Input AH = 45H

DX = Handle

Output AH = Error status:

AH=00H: O.K.

AH=80H: Internal error, EMM possibly destroyed

AH=81H: EMS hardware error

AH=83H: Invalid handle

AH=85H: Error while saving and restoring mapping

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The handle used when calling this function must have been returned by a previous call to EMM function 43H.

All the pages allocated to this handle are released by this function. It is impossible to release individual pages.

After a successful call to this function the handle is no longer valid and cannot be used for accessing EMS memory.

If the function returns an error, you should repeat the call at least three times or the pages will remain allocated and will not be available for other programs.

Interrupt 67H, Function 46H	LIM/EMS
Expanded memory: Get EMM version	

Determines the version number of the EMM (Expanded Memory Manager).

Input AH = 46H

Output AH = 0: O.K.

AL = EMM version number

AH > 0: Error

AH=80H: Internal error, EMM possibly destroyed

AH=81H: EMS hardware error

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The EMM version number is stored in the AL register as a BCD number, in which the upper four bits represent the version number preceding the decimal point and the lower four bits represent the version number following the decimal point. See also the demonstration programs in Chapter 12.

Interrupt 67H, Function 47H	LIM/EMS
Expanded memory: Save mapping	

Saves current mapping between the four physical pages in the page frame and the associated logical pages.

Input	AH = 47H DX = Handle
Output	AH = Error status AH=00H: O.K. AH=80H: Internal error, EMM possibly destroyed AH=81H: EMS hardware error AH=83H: Invalid handle AH=8CH: Mapping memory full AH=8DH: Mapping for handle already stored, not restored using function 48H

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The handle used when calling this function must have been returned by a previous call to EMM function 43H.

This function is intended for use within a TSR program or by the operating system in a multitasking environment, but can be used by any program.

Interrupt 67H, Function 48H	LIM/EMS
Expanded memory: Restore mapping	

Restores mapping between the logical and physical pages saved by function 8H.

Input	AH = 48H DX = handle
Output	AH = Error status: AH=00H: O.K. AH=80H: Internal error, EMM possibly destroyed AH=81H: EMS hardware error AH=83H: Invalid handle AH=8EH: Mapping storage contains no entry for this handle

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The handle used when calling this function must have been returned by a previous call to EMM function 43H.

Calling this function fails whenever the mapping for this handle has not been saved with function 47H, or the mapping has already been restored by a previous call to function 48H.

This function is intended for use within a TSR program or by the operating system in a multitasking environment, but can be used by any program.

Interrupt 67H, Function 49H	LIM/EMS
Expanded memory: Undocumented	
Interrupt 67H, Function 4AH	LIM/EMS
Expanded memory: Undocumented	
Interrupt 67H, Function 4BH	LIM/EMS
Expanded memory: Get number of handles	

Returns the number of memory blocks and the number of handles allocated by function 43H.

Input

AH = 4BH

Output

AH = 0: O.K.

BX = Number of allocated handles

AH > 0: Error

AH=80H: Internal error, EMM possibly destroyed

AH=81H: EMS hardware error

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The number of allocated handles is not the same as the number of programs which are currently accessing the EMS memory. Each program can request an arbitrary number of EMS memory blocks/handles with function 4H.

Interrupt 67H, Function 4CH	LIM/EMS
Expanded memory: Get number of allocated pages	

Returns the number of pages which have been allocated to the specified handle.

Input

AH = 4CH

DX = Handle

Output

AH = 0: O.K.

BX = Number of allocated pages

AH > 0: Error

AH=80H: Internal error, EMM possibly destroyed

AH=81H: EMS hardware error

AH=83H: Invalid handle

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

The number of allocated pages must range from 1 to 512.

Interrupt 67H, Function 4DH**LIM/EMS**

Expanded memory: Get all handles

Loads the numbers of all active handles and the number of pages allocated to each into an array.

Input

AH = 4DH

ES = Segment address of array

DI = Offset address of array

Output

AH = 0: O.K.

BX = Number of allocated logical pages

AH > 0: Error

AH=80H: Internal error, EMM possibly destroyed

AH=81H: EMS hardware error

Remarks

Do not call this function unless you know that EMS memory and a corresponding EMM are installed (see Chapter 12 for more information).

If the function returns successfully, the memory area to which the ES:DI register pair points will contain two words for each active handle. The first word contains the handle itself and the second word contains the number of pages allocated to the handle. The number of these entries is returned in the BX register.

Since the EMM can manage a maximum of 256 handles, the array will never occupy more than 1024 bytes (1K).

Interrupt 67H, Function 4EH, Subfunction 00H**LIM/EMS (Version 3.2 and above)**

Expanded memory: Get page map

Gets page map (a map of logical and physical EMS pages).

Input AH = 4EH
 AL = 00H
 ES:DI = Pointer to empty array

Output AH = 0: O.K.
 ES:DI = Pointer to filled array
 AH >0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

This function requires no EMM handle (compare with function 48H).

Interrupt 67H, Function 4EH, Subfunction 01H	LIM/EMS (Version 3.2 and above)
Expanded memory: Set page map	

Sets the page map (a map of logical and physical EMS pages) to the status that existed before calling subfunction 00H.

Input AH = 4EH
 AL = 01H
 ES:DI = Pointer to page map array

Output AH = 0: O.K.
 AH >0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

This function requires no EMM handle (compare with function 48H).

Interrupt 67H, Function 4EH, Subfunction 02H	LIM/EMS (Version 3.2 and above)
Expanded memory: Swap page map	

Swaps the current page map with the previously stored page map.

Input AH = 4EH
 AL = 02H
 DS:SI = Pointer to the array containing the page map to be restored
 ES:DI = Pointer to the array containing the current page map

Output AH = 0: O.K.

AH >0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Before calling this function, the buffer containing the page map must have been previously loaded using subfunction 00H or subfunction 02H.

Subfunction 03H returns the size of the current page map array.

This function requires no EMM handle (compare with functions 47H and 48H).

Interrupt 67H, Function 4EH, Subfunction 03H

LIM/EMS (Version 3.2 and above)

Expanded memory: Get page map array size

Returns the amount of memory required for the page map.

Input AH = 4EH

AL = 03H

Output AH = 0: OK

AL=Page map array size in bytes

AH >0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Interrupt 67H, Function 4FH, Subfunction 00H

LIM/EMS (Version 4.0 and above)

Expanded memory: Save partial page map

Saves part of a page map.

Input AH = 4FH

AL = 00H

DS:SI = Pointer to map list

ES:DI = Pointer map state buffer

Output AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The map list must contain the number of pages in its first word, followed by a word containing the segment address of these pages. The size of this map list is therefore:

$2 + (\text{number_of_pages_to_be_stored} * 2)$ bytes

Subfunction 02H of this function determines the buffer size needed for saving the selected entries from the page map.

Interrupt 67H, Function 4FH, Subfunction 01H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Restore partial page map

Restores a page map previously saved using function 4FH, subfunction 00H.

Input

AH = 4FH

AL = 01H

DS:SI = Pointer to the buffer with the stored page table

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Before calling this function, the buffer must be saved using function 4FH, subfunction 00H.

Interrupt 67H, Function 4FH, Subfunction 02H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Get partial page map size info

Determines the size of the buffer required for saving part of a page map.

Input

AH = 4FH

AL = 02H

BX = Number of physical pages to be stored

Output

AH = 0: O.K.

AL=Array size in bytes

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Interrupt 67H, Function 50H, Subfunction 00H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Map multiple pages by number

Converts logical pages to physical pages, addressable through a numeric sequence.

Input

AH = 50H

AL = 00H

CX = Number of logical pages

DX = EMM handle by which pages are allocated

DS:SI = Pointer to buffer containing conversion information

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer passed must contain two sequential words for every conversion. The first word specifies the logical page number, while the second word specifies the physical page number where the logical page will be reproduced.

If the logical page number is -1, the corresponding physical page is deleted and cannot be read or written.

Interrupt 67H, Function 50H, Subfunction 01H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Map multiple pages by address

Converts logical pages to physical pages, addressable through segment addresses.

Input

AH = 50H

AL = 01H

CX = Number of logical pages

DX = EMM handle by which pages are allocated

DS:SI = Pointer to buffer containing conversion information

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer passed must contain two sequential words for every conversion. The first word specifies the logical page number, while the second word specifies the physical page number where the logical page will be reproduced.

If the logical page number is -1, the corresponding physical page is deleted and cannot be read or written.

Function 58H, subfunction 00H lets you determine the segment addresses of the available physical pages.

Interrupt 67H, Function 51H
LIM/EMS (Version 4.0 and above)

Expanded memory: Reallocate pages for handle

Reallocates larger or smaller numbers of logical pages. This number belongs to a handle allocated by function 43H.

Input

AH = 48H

DX = EMM handle by which pages are allocated

BX = New number of pages

Output

AH = 0: O.K.

BX = Number of logical pages using this handle

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

You can reduce the number of pages assigned easily. However, increasing the number of pages allocated depends on the amount of free EMS memory available. Check the AH register for errors after every call to this function.

If this function reduces the number of allocated pages, a corresponding number of pages at the upper end of the pages allocated up to now are cut, destroying the contents of those pages.

The handle remains valid after the call, even if the number of pages is reduced to zero.

Interrupt 67H, Function 52H, Subfunction 00H
LIM/EMS (Version 4.0 and above)

Expanded memory: Get handle attribute

Indicates whether EMS pages assigned to a handle are volatile (will not survive in memory after a warm boot) or volatile (will survive a warm boot).

Input

AH = 52H

AL = 00H

DX = EMM page handle

Output

AH = 0: O.K.

AL=Page attribute

0 : Volatile pages

1 : Non-volatile pages

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Interrupt 67H, Function 52H, Subfunction 01H**LIM/EMS (Version 4.0 and above)**

Expanded memory: Set handle attribute

If an EMS-card has a capability to protect EMS-pages against a Warmstart of the computer, the corresponding attribute can be determined with the help of this function.

Input

AH = 52H

AL = 01H

BL = Page attribute

0 : Volatile pages

1 : Non-volatile pages

DX = EMM page handle

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Only a few EMS cards can protect EMS pages from overwriting during a warm boot, and no EMS emulators can perform this task. Before calling this function, select subfunction 02H to determine the availability of non-volatile EMS pages.

Interrupt 67H, Function 52H, Subfunction 02H**LIM/EMS (Version 4.0 and above)**

Expanded memory: Get attribute capability

Determines whether the EMS card can offer protection to EMS pages from a warm boot. This function dictates whether function 52H, subfunctions 00H and 01H can be used.

Input

AH = 52H

AL = 02H

Output

AH = 0: O.K.

AL = 0 : Non-volatile pages not supported

AL = 1: Non-volatile pages supported

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Only a few EMS cards can protect EMS pages from overwriting during a warm boot, and no EMS emulators can perform this task. Call this function to determine the availability of non-volatile EMS pages.

Interrupt 67H, Function 53H, Subfunction 00H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Get handle name

Stores a handle name in the caller's buffer.

Input

AH = 53H

AL = 00H

DX = EMM page handle

ES:DI = Pointer to name buffer

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The name of a handle is always 8 characters in length. For this reason the buffer must have at least 8 bytes of storage capacity.

After function 43H allocates a handle, this new handle contains a 0 as its name. Before calling function 53H, subfunction 00H, call subfunction 01H to attach the name to a handle.

Interrupt 67H, Function 53H, Subfunction 01H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Set handle name

Gives a name to a previously allocated handle for access.

Input

AH = 53H

AL = 01H

DX = EMM page handle

ES:DI = Pointer to name buffer

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The name of a handle is always 8 characters in length. For this reason the buffer must have at least 8 bytes of storage capacity.

Interrupt 67H, Function 54H, Subfunction 00H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Get all handle names

Returns all handle names to a buffer in the caller.

Input

AH = 54H

AL = 00H

ES:DI = Pointer to name buffer

Output

AH = 0: O.K.

AL = Number of active handles

ES:DI = Pointer to filled in name buffer

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer must offer 2,550 bytes of available memory (255 handles administered x 10 bytes for each handle entry). The first two bytes store the actual handle, and the remaining eight bytes contain the handle name.

Interrupt 67H, Function 54H, Subfunction 01H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Search for handle name

Searches for the handle name supplied by the caller.

Input

AH = 54H

AL = 01H

DS:SI = Pointer to name buffer

Output

AH = 0: O.K.

DX = EMM page handle

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Interrupt 67H, Function 54H, Subfunction 02H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Get total handles

Returns total number of handles.

Input AH = 54H

AL = 02H

Output AH = 0: O.K.

BX = Number of handle

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Interrupt 67H, Function 55H, Subfunction 00H

LIM/EMS (Version 4.0 and above)

Expanded memory: Map page by page number/jump

Maps the pages by page numbers, and jumps to one of the pages using a FAR jump.

Input AH = 55H

AL = 00H

DX = EMM page handle

DS:SI = Pointer to buffer

Output AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information). The buffer must have the following structure:

Addr.	Contents	Type
+00H	FAR pointer to the jump target	1 ptr
+05H	Number of pages to map before jump	1 byte
+06H	FAR pointer to map list	1 ptr
Length 9 bytes		

The structure of the map list is identical to the structure of this list during the construction of function 50H, subfunction 00H. The logical page number and its corresponding physical page are retained.

Interrupt 2FH, Code 55H, Subfunction 01H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Map page by page number/call

Maps the pages by page segments, and jumps to one of the pages using a FAR jump.

Input

AH = 55H

AL = 01H

DX = EMM page handle

DS:SI = Pointer to buffer

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer must have the following format:

Addr.	Contents	Type
+00H	FAR pointer to the jump target	1 ptr
+05H	Number of pages to map before jump	1 byte
+06H	FAR pointer to map list	1 ptr
Length 9 bytes		

The format of the map list is identical to the format of this list during the execution of function 50H, subfunction 00H. The logical page number and its corresponding physical page are retained.

Interrupt 67H, Function 56H, Subfunction 01H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Map page by page segment/call

Maps pages by page segments and calls a program through a FAR call. After this routine ends, a second page map occurs, and interrupt function control returns to the caller.

Input

AH = 56H

AL = 01H

DX = EMM page handle

DS:SI = Pointer to the buffer

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer must have the following format:

Addr.	Contents	Type
+00H	FAR pointer to the call's target	1 ptr
+05H	Number of pages to be mapped before call	1 byte
+06H	FAR pointer to list of pages to be mapped before call	1 ptr
+0AH	Number of pages to map before returning	1 byte
+0BH	FAR pointer to list of pages to be mapped before return	1 ptr
Length: 14 byte		

The format of the map list is identical to the format of this list during the execution of function 50H, subfunction 00H. The logical page number and its corresponding physical page are retained.

Call subfunction 02H to ensure that enough memory is available before calling this function.

Interrupt 67H, Function 56H, Subfunction 02H	LIM/EMS (Version 4.0 and above)
Expanded memory: Get stack space for map page/call	

Returns the memory space required by subfunctions 00H and 01H.

Input	AH = 56H
	AL = 02H
Output	AH = 0: O.K.
	BX = Space required
	AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Interrupt 67H, Function 57H, Subfunction 00H	LIM/EMS (Version 4.0 and above)
Expanded memory: Move memory region	

Copies memory regions between expanded memory and conventional memory, or within either set of memory.

Input	AH = 57H
	AL = 00H
	DS:SI = Pointer to buffer

Output AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer must have the following format:

Addr.	Contents	Type
+00H	Region length in bytes	1 dword
+04H	Source memory type (0 =conventional, 1 =EMS)	1 byte
+05H	Source memory handle (EMS only)	1 word
+07H	Source memory offset	1 word
+09H	Source memory segment address (conv. memory) or logical page number (EMS)	1 word
+0BH	Target memory type (0 =conventional, 1 =EMS)	1 byte
+0CH	Target memory handle (EMS only)	1 word
+0EH	Target memory offset	1 word
+10H	Target memory segment address (conv. memory) or logical page number (EMS)	1 word
Length: 18 bytes		

Interrupt 2FH, Code 57H, Subfunction 01H

LIM/EMS (Version 4.0 and above)

Expanded memory: Exchange memory regions

This function operates similar to the subfunction 00H. It does not copy one storage area into another, but swaps the content of the indicated storage areas.

Input AH = 57H

AL = 01H

DS:SI = Pointer to buffer

Output AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer must have the following format:

Addr.	Contents	Type
+00H	Region length in bytes	1 dword
+04H	Source memory type (0 = onventional, 1 =EMS)	1 byte
+05H	Source memory handle (EMS only)	1 word
+07H	Source memory offset	1 word
+09H	Source memory segment address (conv. memory) or logical page number (EMS)	1 word
+0BH	Target memory type (0 =conventional, 1 =EMS)	1 byte
+0CH	Target memory handle (EMS only)	1 word
+0EH	Target memory offset	1 word
+10H	Target memory segment address (conv. memory) or logical page number (EMS)	1 word
Length: 18 bytes		

This function can swap up to 1 megabyte of memory.

If EMS pages are participating in this swap and the size specified exceeds the EMS page, other EMS pages will be used.

The source and destination regions may not overlap.

Interrupt 67H, Function 58H, Subfunction 00H	LIM/EMS (Version 4.0 and above)
Expanded memory: Get addresses of mappable pages	

Returns addresses of all physical EMS pages and related page numbers.

Input

AH = 58H

AL = 00H

ES:DI = Pointer to buffer

Output

AH = 0: O.K.

CX=Number of entries

ES:DI=Pointer to filled in buffer

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Subfunction 01H helps determine the size of the buffer required.

EMM passes the desired information into the buffer in two word entries. The first word contains the segment address of an EMS page, while the second word indicates the page number that applies to the page.

Interrupt 67H, Function 58H, Subfunction 01H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Get number of mappable pages

Determines the number of physical EMS pages, and calculates the size of the buffer required by subfunction 00H.

Input

AH = 58H

AL = 01H

Output

AH = 0: O.K.

CX=Number of pages

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The size of the buffer required is the result of multiplying the number of pages (the contents of the CX register) by four.

Interrupt 67H, Function 59H, Subfunction 00H**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: Get hardware configuration

Determines EMS hardware configuration. This function only applies to an operating system equipped with EMS.

Input

AH = 59H

AL = 00H

ES:DI = Pointer to buffer

Output

AH = 0: O.K.

AL=Page map size in bytes

ES:DI=Pointer to filled in buffer

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

The buffer must be a minimum size of 10 bytes. It contains the following information after the function call:

Addr.	Contents	Type
+00H	Raw EMS page size in paragraphs	1 word
+02H	Alternate EMS register sets	1 word
+04H	Mapping context save area size in bytes	1 word
+06H	Number of assignable register sets	1 word
+08H	DMA operating mode (0 = DMA with alt register sets 1 = One DMA register set)	1 word
Length: 10 bytes		

Interrupt 67H, Function 59H, Subfunction 01H	LIM/EMS (Version 4.0 and above)
Expanded memory: Get number of raw pages	

Determines the total number of raw (non-standard EMS) pages, and indicates which of those pages are available. A raw page has a size other than the default 16K.

Input AH = 59H

AL = 01H

Output AH = 0: O.K.

DX=Total number of raw pages

BX=Number of raw pages not allocated

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information). Not all EMS cards support raw pages. This may result in a value of zero returned in the DX register.

Interrupt 67H, Function 5AH, Subfunction 00H	LIM/EMS (Version 4.0 and above)
Expanded memory: Allocate handle & standard pages	

Allocates standard EMS pages. Unlike function 43H, this function can allocate zero pages without error.

Input AH = 5AH

AL = 0H

BX = Number of standard pages

Output AH = 0: O.K.

DX=EMM page handle

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Interrupt 67H, Function 5AH, Subfunction 01H	LIM/EMS (Version 4.0 and above)
Expanded memory: Allocate handle & raw pages	

Allocates raw pages. A raw page has a size other than the default 16K.

Input

AH = 5AH

AL = 01H

BX = Number of pages to be allocated

Output

AH = 0: O.K.

DX=EMM page handle

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

Not all EMS cards support raw pages. Test the function status in the AH register.

This function can also allocate zero pages.

Interrupt 67H, Function 5BH	LIM/EMS (Version 4.0 and above)
Expanded memory: Alternate map and DMA services	

Function 5BH contains nine subfunctions used exclusively in operating system development. None of these subfunctions are used in normal PC software development.

Interrupt 67H, Function 5CH	LIM/EMS (Version 4.0 and above)
Expanded memory: Prepare EMM for warm boot	

Prepares the Expanded Memory Manager for an imminent warm boot of the computer. This gives the EMM an opportunity to store internal data and prevent the loss of non-volatile pages when the warm boot occurs.

Input

AH = 5CH

Output

AH = 0: O.K.

AH > 0: Error

Remarks

Do not call this function unless you know that EMS memory and a corresponding version of EMM are installed (see Chapter 12 for more information).

See also function 52H for more information.

Interrupt 67H, Function 5DH**LIM/EMS** (*Version 4.0 and above*)

Expanded memory: EMM operating system services

Function 5BH contains three subfunctions used exclusively in operating system development. None of these subfunctions are used in normal PC software development.

XMS Functions

The XMS (eXtended Memory Specification) was created by Microsoft in cooperation with several other corporations, as a supplement to the EMS standard. This specification coordinates access to extended memory, the memory which lies beyond the 1 megabyte limit of conventional memory.

Interrupt 2FH must recognize an XMS driver, and the XMS handler's address must be specified, before calling any of these functions. XMS functions are called through a FAR CALL instruction, instead of the special interrupt used by other function interfaces. See Chapter 12 for more information.

Error codes are placed in the BL register. The following error codes can occur during XMS function calls:

80H	Function not implemented	A4H	Source offset is invalid
81H	VDISK device driver was detected	A5H	Destination handle is invalid
82H	A20 error	A6H	Destination offset is invalid
8EH	General driver error	A7H	Length is invalid
8FH	Unrecoverable driver error	A8H	Overlap in move request is invalid
90H	HMA does not exist	A9H	Parity error detected
91H	HMA is already in use	AAH	Block not locked
92H	DX is less than /HMAMIN = parameter	ABH	Block locked
93H	HMA is not allocated	ACH	UMB count overflowed
94H	A20 line is still enabled	ADH	Lock failed
A0H	All extended memory is allocated	B0H	Smaller UMB is available
A1H	EMM handles are exhausted	B1H	No UMBs are available
A2H	Handle is invalid	B2H	UMB segment number is invalid
A3H	Source handle is invalid		

Function 00H	XMS
Determine XMS version number	

Returns the XMS driver's version number and the driver's internal revision number.

Input	AH = 00H
Output	AX = XMS version number BX = Internal revision number

DX = HMA status
 0: No HMA
 1: HMA available

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

Function 01H	XMS
Allocate High Memory Area (HMA)	

Allocate High Memory Area (HMA)

Reserves all or part of the HMA for program use.

Input AH = 01H
 DX = Requested HMA space in bytes
 FFFFH: Amount needed for application program
 < FFFFH: Actual amount needed for operating system or driver

Output AX = 0001H: Operation completed
 AX = 0000H: Error
 BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

Address line A20 must be freed using function 03H or function 05H, before HMA access can occur.

This function can fail if a TSR doesn't request the entire HMA, and if the amount of memory requested is less than the value passed by the /HMAMIN parameter during driver installation. These factors give a TSR, which may require only a few kilobytes, exclusive access rights to the HMA, while prohibiting access by another program requiring more memory.

If the HMA should not remain in possession of the program beyond its termination, it must be freed before the end of program execution by function 02H.

Function 02H	XMS
Free High Memory Area (HMA)	

Releases the HMA previously allocated by function 01H.

Input AH = 02H

Output AX = 0001H: Operation completed
 AX = 0000H: Error

BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

A previously allocated HMA remains allocated even after program execution ends. This function must be called before the HMA can be allocated by other subsequent program calls.

Calling function 02H destroys the HMA's current contents.

Function 03H

XMS

Globally enable address line A20

Globally enables address line A20, permitting direct HMA access when the processor is in real mode.

Input

AH = 03H

Output

AX = 0001H: Operation completed

AX = 0000H: Error

BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

This function must be called before or after allocating the HMA. This ensures HMA access in real mode through segment address FFFFH.

Function 04H should be used to disable address line A20 before program execution ends. This prevents a possible segment overflow in subsequent program calls.

The process of enabling address line A20 can take much time. We recommend that you call function 03H only when necessary.

Function 04H

XMS

Globally disable address line A20

Globally disables address line A20, prohibiting direct HMA access when the processor is in real mode.

Input

AH = 04H

Output

AX = 0001H: Operation completed

AX = 0000H: Error

BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

The process of disabling address line A20 can take much time. We recommend that you call function 04H only when necessary.

Function 05H	XMS
Locally enable address line A20	

Locally enables address line A20. Calls to functions 05H and 06H are recorded internally with the help of a counter, which increments with every call to function 05H and decrements with every call to function 06H. This line can be enabled only if it is currently disabled.

Input	AH = 05
Output	AX = 0001H: Operation completed
	AX = 0000H: Error
	BL = Error code (see below)

Remarks

This function must be called before or after allocating the HMA.

Function 06H should be used to disable address line A20 before program execution ends. This prevents a possible segment overflow in subsequent program calls.

The process of enabling address line A20 can take much time. We recommend that you call function 05H only when necessary.

Function 06H	XMS
Locally disable address line A20	

Locally disable address line A20, if line A20 was previously enabled by function 05H.

Input	AH = 06
Output	AX = 0001H: Operation completed
	AX = 0000H: Error
	BL = Error code (see below)

Remarks

Calls to functions 05H and 06H are recorded internally with the help of a counter, which increments with every call to function 05H and decrements with every call to function 06H. Only after reaching the value 0 is the address line actually disabled. Every call to function 06H must be preceded by a call to function 05H.

Function 06H should be used to disable address line A20 before program execution ends. This prevents a possible segment overflow in subsequent program calls.

The process of enabling address line A20 can take much time. We recommend that you call function 05H only when necessary.

Function 07H	XMS
Query status of address line A20	

Determines whether address line A20 has been enabled by functions 03H or 05H.

Input AH = 07H

Output AX = 0001H: Address line A20 enabled

AX = 0000H: Address line A20 disabled

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

This query is hardware-controlled, and takes less time than the process of enabling or disabling address line A20.

Function 08H	XMS
Query free extended memory	

Returns the total amount of free extended memory, as well as the size of the largest free block of extended memory.

Input AH = 08H

Output AX = Size of the largest free block of extended memory in kilobytes

DX = Total free extended memory in kilobytes

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

The HMA is excluded from this function, even if no HMA has been allocated. This results in a total 64K too large, because extended memory allocation always starts after the HMA. See Chapter 12 for more information.

Function 09H	XMS
Allocate Extended Memory Block (EMB)	

Allocates an EMB for program use.

Input AH = 09H

DX = Size of the requested block in kilobytes

Output AX = 0001H: Operation completed

AX = 0000H: Error

BL = Error code (see below)

DX = Handle for additional EMB access, if additional access is possible

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

The returned handle accesses the EMB during all subsequent calls, and must therefore be stored in the program. If the program loses the handle, the user can only free the EMB by resetting the computer.

The XMS can only honor an EMB allocation if a handle is free, and if a large enough memory block is available. Since the number of handles is limited (usually 32), large EMBs should be allocated to avoid calling the function too often, and to minimize handle calls.

The allocated EMB must be freed using 0AH before program execution ends, or the EMB will be lost for passing to subsequent applications.

Function 0AH	XMS
Free allocated Extended Memory Block (EMB)	

Frees an Extended Memory Block (EMB) previously allocated using function 09H.

Input	AH = 0AH
	DX = Handle
Output	AX = 0001H: Operation completed
	AX = 0000H: Error
	BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

Calling function 0AH destroys the EMB's current contents and invalidates the handle.

Function 0BH	XMS
Move Extended Memory Block (EMB)	

Transfers memory between conventional RAM and extended memory, or copies blocks within conventional RAM or extended memory.

Input	AH = 0BH
	DS:SI = Pointer to the following structure which determines the memory area to be copied and its destination.
Output	AX = 0001H: Operation completed
	AX = 0000H: Error
	BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

A handle value of 0 indicates access to conventional RAM. The segment and offset addresses specify the beginning of the memory block.

When addressing an EMB, the handle returned after allocating the EMB using function 09H must be indicated. The offset address then represents the offset relative to the start of the block.

The handles specified cannot be handles whose corresponding EMB was locked using function 0CH.

If the indicated blocks overlap, the source block must precede the destination block, or the function may fail.

The source block copies faster if both blocks begin at even numbered addresses in an AT, or if both blocks begin at addresses divisible by four in an 80386.

The Extended Memory Move Structure		
Addr	Content	Type
+00H	Block length in bytes (must be an even number)	1 DWORD
+04H	Handle of the source Block	1 WORD
+06H	Source block offset, where copying starts	1 DWORD
+0AH	Handle of the destination Block	1 WORD
+0CH	Destination block offset, where copying starts	1 DWORD
Length: 16 bytes		

Function 0CH	XMS
Lock Extended Memory Block (EMB)	

Locks an Extended Memory Block (EMB) to a specific memory location. The XMM moves EMBs to different locations after the release of an EMB, thus preventing memory gaps. Function 0CH ensures that the specified EMB remains in the same location in memory.

Input AH = 0CH

DX = EMB handle

Output AX = 0001H: Operation completed

AX = 0000H: Error

BL = Error code (see below)

DX:BX = Linear 32-bit address of the EMB in memory

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

The function call returns the EMB's address. This address is valid until the EMB is freed.

Release locked blocks as soon as possible using function 0DH. This minimizes any hindering of the XMM's tasks.

Calls to functions 0CH and 0DH are recorded internally with the help of a counter, which increments with every call to function 0CH and decrements with every call to function 0DH. Only after reaching the value 0 can the corresponding EMB be unlocked.

Function 0DH	XMS
Unlock Extended Memory Block (EMB)	

Unlocks an Extended Memory Block (EMB) previously locked using function 0DH. The XMM moves EMBs to different locations after the release of an EMB, thus preventing memory gaps. Function 0DH ensures that the specified EMB is also available to the XMM.

Input	AH = 0DH
	DX = EMB handle
Output	AX = 0001H: Operation completed
	AX = 0000H: Error
	BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

This function invalidates the EMB address returned during the call of function 0CH.

Calls to functions 0CH and 0DH are recorded internally with the help of a counter, which increments with every call to function 0CH and decrements with every call to function 0DH. Only after reaching the value 0 can the corresponding EMB be unlocked.

Function 0EH	XMS
Get (EMB) handle information	

Provides information about an EMB. This includes the block size, its block counter and the number of handles free.

Input	AH = 0EH
	DX = EMB handle
Output	AX = 0001H: Operation completed
	BH = Block counter
	BL = Number of free EMB handles
	DX = EMB length in kilobytes
	DX = 0000H: Error
	BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

Calls to functions 0CH and 0DH are recorded internally with the help of a block counter, which increments with every call to function 0CH and decrements with every call to function 0DH. A value unequal to zero indicates the number of calls which must be made to 0DH before the EMB can be unlocked.

Function 0FH	XMS
Resize Extended Memory Block (EMB)	

Allows the enlargement or reduction of the size of an EMB.

Input	AH = 0FH
	BX = New size in kilobytes
	DX = EMB handle
Output	AX = 0001H: Operation completed
	AX = 0000H: Error
	BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

The indicated EMB must not be locked.

If you reduce the size of the EMB, the contents of the upper end of the EMB are lost.

Function 10H	XMS
Allocate Upper Memory Block (UMB)	

Allocates an upper memory block in the RAM existing between the 640K limit and the beginning of extended memory.

Input	AH = 10H
	DX = Size of the requested memory blocks in paragraphs
Output	AX = 0001H: Operation completed
	BX = Segment address of the UMB
	BX = 0000H: Error
	BL = Error code (see below)
	DX = Maximum size of an allocatable UMB in paragraphs

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

This function is extremely hardware dependent and not implemented in all XMS drivers. After its call the error status must be checked to ensure that a UMB actually was allocated.

Direct UMB access is possible in real mode with the returned segment address. During this access, the offset address derived from the conversion of block length may not be exceeded.

This function can also be used to determine the largest available UMB, by using an unrealistic value (for example FFFFH) for the size of the requested block. No UMB is allocated, but the function returns the length of the largest available UMB.

Function 11H	XMS
Free allocated Upper Memory Block (UMB)	

Frees a UMB previously allocated using function 10H, making the UMB available to other programs.

Input

AH = 11H

DX = UMB segment address

Output

AX = 0001H: Operation completed

AX = 0000H: Error

BL = Error code (see below)

Remarks

Interrupt 2FH must recognize an XMS driver, and the XMS driver jump location must be specified, before calling this function.

After calling function 11H, the UMB's contents are lost. No memory access can be made through the block's segment address.

All allocated UMBs should be released using this function before program execution ends. Otherwise, it may not be possible to pass the UMBs to subsequent programs.

Mouse Driver Functions

Microsoft has supported its mouse with a software-interface in the form of the MOUSE.SYS device driver or the MOUSE.COM program. This interface has been established and is imitated by all other mouse manufacturers. This insures full compatibility with the Microsoft mouse. The various functions of the mouse-interface are called through Interrupt 33h, where a 16 bit value is used as function number, which must be passed in the AX-Register.

Version 8.0 of the mouse driver supports 53 different functions, with function numbers from 0000h to 00034h. However, the functions 11h and 12h, as well as 002Eh are undocumented and therefore are only used inside the mouse driver.

The Microsoft mouse-driver has been subjected to many revisions and enhancements. Therefore numerous variants exist between the version 1.0 and 8.0. Versions with a version number smaller than 6.26 are rarely still in circulation, because they do not work with the latest video cards.

Unfortunately, Microsoft reveals the date of the introduction of the various mouse driver functions only since the function 25h, which was introduced with version 6.26. Many of preceding functions existed since version 1.0, but many were added during the course of development between version 1.0 and 6.26. Since more accurate information is lacking, it will be assumed in the framework of this reference, that these functions have been available since version 1.0. Even if this is not true in a particular case, you can assume that these functions are supported by all mouse-drivers which are in use.

Most mouse functions operate with the AX, BX, CX and DX, registers to pass information from the caller and to return function results. Only in exceptional cases are the ES, SI and DS registers used for the function call, usually when the address of buffers must be passed.

Basically only the content of the register changes for the return of the function results. The content of all other registers remains unchanged. For most functions, the AX-Register is used for the return of the function status, which in case of error indicates the failure of the operation.

Interrupt 33H, Function 00H	Mouse
Resets (initializes) the mouse driver	

Resets (initializes) the mouse driver.

Input	AX = 00H
Output	AX = Mouse installation status
	AX=FFFFH: Mouse driver installed
	AX=0000H: Error, no mouse driver installed
	BX = Number of mouse buttons

Remarks

The reset process executes the following tasks:

Moves the mouse cursor; to the center of the screen and clears the cursor from the screen. When enabled, the default cursor appears as an inverse video square. The representation is always in display page 0, independent of the current display mode. The entire screen area becomes the total range of mouse movement.

Installs the event handler is installed by a program (default is disabled).

Installs light pen emulation (default is disabled).

Specifies mouse cursor's speed. Default relative speed is 8 mickeys per 8 horizontal pixels and 16 mickeys per 16 vertical pixels.

Specifies maximum mouse speed (default is 64 mickeys per second).

Interrupt 33H, Function 01H	Mouse
Display mouse cursor	

Displays the mouse cursor on the screen. This cursor follows any movement the user makes with the mouse device.

Input AX = 01H

Output No output

Remarks

This function increments an internal counter which determines whether the mouse cursor should be displayed on the screen. When the mouse driver is initialized using function 00H, this cursor contains the value -1 (i.e., the mouse cursor does not appear). If this counter contains the value 0 after calling function 01H, the mouse cursor appears on the screen.

The mouse driver follows the mouse movement even when the mouse cursor is not displayed on the screen. After calling this function, the mouse cursor may not appear at the same location as it was when the cursor was previously removed by calling function 00H or function 02H.

Interrupt 33H, Function 02H	Mouse
Hide mouse cursor	

Removes the mouse cursor from the screen.

Input AX = 02H

Output No output

Remarks

This function decrements an internal counter which determines whether the mouse cursor should appear on the screen. If the counter contains the value 0, the mouse cursor is displayed on the screen, while the value -1 removes the mouse cursor from the screen.

The mouse driver follows the mouse movement even when the mouse cursor is not displayed on the screen.

After calling this function, the mouse cursor may not appear at the same location as it was when the cursor was previously removed by calling function 00H or function 02H.

Interrupt 33H, Function 03H	Mouse
Get cursor position/button status	

Returns the current position of the mouse cursor and the current status of the mouse buttons.

Input AX = 03H

Output BX = Mouse button status

Bit 0=1: Left mouse button activated

Bit 1=1: Right mouse button activated

Bit 2=1: Center mouse button activated

Bits 3-15: Unused

CX = X coordinate (horizontal mouse position)

DX = Y coordinate (vertical mouse position)

Remarks

The coordinates returned in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

If the mouse is equipped with only two mouse buttons, the information about the central mouse button does not have significance.

Interrupt 33H, Function 04H	Mouse
Move mouse cursor	

Moves the active mouse cursor to a certain position on the screen.

Input AX = 04H

CX = X coordinate (horizontal mouse position)

DX = Y coordinate (vertical mouse position)

Output No output

Remarks

The coordinates returned in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

If the position indicated is outside the range of movement specified by functions 07H and 08H, the function adjusts coordinates so that the mouse cursor remains within this range of movement.

The mouse cursor moves to the new position, even if the mouse is not currently visible. Once re-enabled, the mouse cursor appears at this new position.

Interrupt 33H, Function 05H	Mouse
Determine number of times mouse button was activated	

Informs the calling program of how often a mouse button has been pressed since the last call of function 05H. Function 05H also informs the calling program of the cursor's location on the screen when the button was last activated.

Input

AX = 05H

BX = Mouse button activated

BX=0: Left mouse button

BX=1: Right mouse button

BX=2: Center mouse button

Output

BX = Status of all mouse buttons:

Bit 0=1: Left mouse button activated

Bit 1=1: Right mouse button activated

Bit 2=1: Center mouse button activated

Bits 3-15: Unused

BX = Mouse buttons activated since last function call

CX = Horizontal mouse position during the last activation

DX = Vertical mouse position during the last activation

Remarks

The coordinates returned in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen. The activation counter for the mouse button addressed is reset to 0 when this function is called.

Interrupt 33H, Function 06H	Mouse
Determine number of times mouse button was released	

Informs the calling program of how often a mouse button has been released since the last call of function 06H. Function 06H also informs the calling program of the cursor's location on the screen when the button was last activated.

Input

AX = 06H

BX = mouse button addressed

BX=0: Left mouse button

BX=1: Right mouse button

BX=2: Center mouse button

Output	BX = Status of all mouse buttons
	Bit 0=1: Left mouse button activated
	Bit 1=1: Right mouse button activated
	Bit 2=1: Center mouse button activated
	Bits 3-15: Unused
	BX = Mouse buttons activated since last function call
	CX = Horizontal mouse position during the last activation
	DX = Vertical mouse position during the last activation

Remarks

The coordinates returned in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

The activation counter for the mouse button addressed is reset to 0 when this function is called.

Interrupt 33H, Function 07H	Mouse
Set horizontal range of movement	

Defines the horizontal range of movement for the mouse cursor. Once set, the user cannot move the mouse cursor out of this range.

Input	AX = 07H
	CX = Minimal horizontal cursor position
	DX = Maximum horizontal cursor position

Output	No output
---------------	-----------

Remarks

The coordinates passed in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

If the mouse cursor is outside of this range when function 07H is called, the mouse driver automatically moves the mouse cursor within the limits of the range of movement. If the value in the DX register is less than the value in the CX registers, the two parameters are exchanged.

Interrupt 33H, Function 08H	Mouse
Set vertical range of movement	

Defines the vertical movement range for the mouse cursor. When set, the user cannot move the mouse cursor out of this range.

Input	AX = 08H
	CX = Minimum vertical cursor position
	DX = Maximum vertical cursor position

Output No output

Remarks

The coordinates passed in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

If the mouse cursor is outside of this range when function 07H is called, the mouse driver automatically moves the mouse cursor within the limits of the range of movement.

If the value in the DX register is less than the value in the CX registers, the two parameters are exchanged.

Interrupt 33H, Function 09H	Mouse
Set mouse cursor (graphic mode)	

Defines the appearance of the mouse cursor in graphic mode, as well as the ..bitfield; which compensates for the pixels around the mouse cursor.

Input

AX = 09H

BX = Cursor width starting at left border of bitfield

CX = Cursor height starting at top border of bitfield

ES = Segment address of bitfield

DX = Offset address of bitfield

Output No output

Remarks

The bitfield consists of 64 bytes, of which the first 32 are an AND comparison, and the remaining 32 are an OR combination. Both sets of bytes are based upon the current pixel pattern.

Interrupt 33H, Function 0AH	Mouse
Set mouse cursor (text mode)	

Defines the bitmask which specifies the appearance of the mouse cursor in text mode.

Input

AX = 0AH

BX = Cursor type

BX=0: Software cursor

BX=1: Hardware cursor

CX = AND mask (software cursor) or starting line (hardware cursor)

DX = XOR mask (software cursor) or ending line (hardware cursor)

Output No output

Remarks

If the software cursor is selected, the code of the character beneath the mouse cursor and its attribute byte are combined logically with the mask in the CX register through a binary AND, and then with the value in the DX register through an exclusive OR (XOR). The attribute byte is combined with the most significant byte (CH and DH). The character code is combined with the least significant byte (CL and DL).

The hardware cursor is the same shape as the normal text mode cursor. Monochrome mode values for the starting and ending lines range from 0 to 13. Color mode values for the starting and ending lines range from 0 to 7.

Interrupt 33H, Function 0BH	Mouse
Determine movement values	

Determines the distance between the current mouse position and the mouse position during the last call of function 0BH.

Input	AX = 0BH
Output	CX = Horizontal distance from last point in mickeys DX = Vertical distance from last point in mickeys

Remarks

These values must be interpreted as signed numbers. Positive values indicate movement toward the bottom or right border of the screen, while negative values indicate movement toward the top or left border of the screen.

These values are given in mickeys.(1 mickey=1/200 inch) rather than in pixels.

Interrupt 33H, Function 0CH	Mouse
Set event handler	

Sets the address of an event handler called by the mouse driver when a particular mouse event occurs.

Input	AX = 0CH
	CX = Events which trigger the call of the event handler (event mask)
	Bit 0: Mouse movement
	Bit 1: Left mouse button activated
	Bit 2: Left mouse button released
	Bit 3: Right mouse button activated
	Bit 4: Right mouse button released
	Bit 5: Center mouse button activated
	Bit 6: Center mouse button released
	Bits 7-15: Unused
	ES = Segment address of handler
	DX = Offset address of handler

Output No output

Remarks

The event handler is called by the mouse driver through a FAR call assembler instruction, and therefore must be terminated with a FAR RET instruction. None of the various processor registers may be returned to the caller with a changed content.

The mouse driver passes the following information to the event handler through the processor registers during the call:

- AX = Event mask. The bits correspond to the various events as indicated in the CX register during the installation of the event handler. In addition, other bits can be set, since the value reflects the current status of the mouse driver, and is not limited to the selected events.
- BX = Mouse button status:
 - Bit 0 = Left mouse button activated
 - Bit 1 = Right mouse button activated
 - Bit 2 = Center mouse button activated
- CX = Horizontal mouse position.
- DX = Vertical mouse position.
- SI = Length of last horizontal mouse movement.
- DI = Length of the last vertical mouse movement.
- DS = Data segment of the mouse driver.

The coordinates returned in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

The values in the SI and DI registers refer to mickeys (one mickey = 1/200 inch).

These mickey values must be interpreted as signed numbers. Positive values indicate movement toward the bottom or right border of the screen, while negative values indicate movement toward the top or left border of the screen.

Interrupt 33H, Function 0DH	Mouse
Enable light pen emulation	

Enables emulation of the light pen, and simulates a light pen which if none is present.

Input AX = 0DH

Output No output

Remarks

Light pen emulation only makes sense when used with an application which supports the light pen, or makes light pen reading routines available (e.g., the PEN command in PC-BASIC).

The light pen and mouse are closely related in programming: The position of the mouse cursor is directly related to the light pen's position on the screen, and pressing the left and right mouse button has the same result as pressing the button on the light pen.

Interrupt 33H, Function 0EH	Mouse
Disable light pen emulation	

Disables the light pen emulation enabled by a previous call to function 0DH.

Input AX = 0EH

Output No output

Remarks

Light pen emulation only makes sense when used with an application which supports the light pen, or makes light pen reading routines available (e.g., the PEN command in PC-BASIC).

The light pen and mouse are closely related in programming: The position of the mouse cursor is directly related to the light pen's position on the screen, and pressing the left and right mouse button has the same result as pressing the button on the light pen.

Interrupt 33H, Function 0FH	Mouse
Set cursor speed	

Defines the relationship between mickeys and screen pixels. This specifies the sensitivity of the mouse and the speed at which the mouse cursor moves across the screen.

Input AX = 0FH

 CX = Number of horizontal mickeys

 DX = Number of vertical mickeys

Output No output

Remarks

Values in the CX and DX registers can range from 1 to 32767.

The default setting is 8 horizontal mickeys and 16 vertical mickeys. This causes the mouse cursor to move twice as fast horizontally as it moves vertically. Calling function 00H (Reset mouse driver) changes any previously set values to the default values.

Interrupt 33H, Function 10H	Mouse
Exclusion area	

Designates any area of the screen as an exclusion area. The mouse cursor disappears if moved into the exclusion area.

Input AX = 10H

 CX = X-coordinate, upper left corner of exclusion area

 DX = Y-coordinate, upper left corner of exclusion area

 SI = X-coordinate, lower right corner of exclusion area

 DI = Y-coordinate, lower right corner of exclusion area

Output No output

Remarks

The coordinates passed in the CX, DX, DI and SI registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

Calling function 00H (Reset mouse driver) or function 01H (Display mouse cursor) deletes the exclusion area coordinates.

Interrupt 33H, Function 11H	Mouse
Undocumented	

This undocumented function is used exclusively by the mouse driver, and cannot be called from a program.

Interrupt 33H, Function 12H	Mouse
Undocumented	

This undocumented function is used exclusively by the mouse driver, and cannot be called from a program.

Interrupt 33H, Function 13H	Mouse
Set maximum for mouse speed doubling	

Sets the maximum limit for doubling mouse speed. If the speed of the mouse movement exceeds a certain limit, the mouse driver doubles the mouse cursor speed by doubling the movement's relationship between points and mickeys.

Input AX = 13H
DX = Limit in mickeys per second

Output No output

Remarks

1 mickey=1/200 inches.

To prevent doubling of the mouse speed, the limit can be set higher.

Speeds in excess of 5,000 mickeys per second cannot be achieved by practical means.

Interrupt 33H, Function 14H	Mouse
Exchange event handlers	

Installs a new event handler for certain mouse events, but also retains the address of the old event handler.

Input AX = 14H
CX = Events which should trigger event handler call
Bit 0: Mouse movement
Bit 1: Left mouse button activated
Bit 2: Left mouse button released

Bit 3: Right mouse button activated

Bit 4: Right mouse button released

Bit 5: Center mouse button activated

Bit 6: Center mouse button released

Bit 7-15: Unused

ES = Segment address of new event handler

DX = Offset address of new event handler

Output

CX = Event mask of the previously installed event handler

ES = Segment address of previously installed event handler

DX = Offset address of previously installed event handler

Remarks

The event handler is called by the mouse driver through a FAR call assembler instruction, and therefore must be terminated with a FAR RET instruction. None of the various processor registers may be returned to the caller with a changed content.

The mouse driver passes the following information to the event handler through the processor registers during the call:

AX = Event mask. The bits correspond to the various events as indicated in the CX register during the installation of the event handler. In addition, other bits can be set, since the value reflects the current status of the mouse driver, and is not limited to the selected events.

BX = Mouse button status:

Bit 0 = Left mouse button activated

Bit 1 = Right mouse button activated

Bit 2 = Center mouse button activated

CX = Horizontal mouse position.

DX = Vertical mouse position.

SI = length of last horizontal mouse movement.

DI = Length of the last vertical mouse movement.

DS = Data segment of the mouse driver.

The coordinates returned in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

The values in the SI and DI registers refer to mickeys (one mickey = 1/200 inch).

These mickey values must be interpreted as signed numbers. Positive values indicate movement toward the bottom or right border of the screen, while negative values indicate movement toward the top or left border of the screen.

Interrupt 33H, Function 15H	Mouse
Determine mouse status buffer size	

Returns the size of the mouse status buffer, in which a program can store the complete status of the mouse driver.

Input AX = 15H

Output BX = Mouse status buffer size in bytes

Remarks

Function 16H (Store mouse status) stores the mouse status in the buffer.

Interrupt 33H, Function 16H	Mouse
Store mouse status	

Stores mouse status information in a buffer.

Input AX = 16H

 ES = Segment address of mouse status buffer

 DX = Offset address of mouse status buffer

Output No output

Remarks

The caller is responsible for creating a buffer large enough to contain all the status information. Before calling this function, call function 15H (Determine mouse status buffer size) to determine the size of the mouse status buffer.

This function works well when called before executing a program using the EXEC function. This allows the mouse status to be saved in memory, then restored from within the called program.

Interrupt 33H, Function 17H	Mouse
Restore mouse status	

Reads all mouse parameters from a buffer stored by function 16H.

Input AX = 17H

 ES = Segment address of mouse status buffer

 DX = Offset address of mouse status buffer

Output No output

Interrupt 33H, Function 18H	Mouse
Install alternate event handler	

This function permits a program to install a limited range event handler. This handler can be called by the mouse driver when certain mouse events occur in conjunction with the keyboard.

Input

AX = 0018H

CX = Events which should trigger the call of the event handler

Bit 0: Mouse movement

Bit 1: Left mouse button activated

Bit 2: Left mouse button released

Bit 3: Right mouse button activated

Bit 4: Right mouse button released

Bit 5: Shift key pressed during mouse button event

Bit 6: Ctrl key pressed during mouse button event

Bit 7: Alt key pressed during mouse button event

Bits 8-15: Unused

ES = Segment address of event handler

DX = Offset address of event handler

Output

AX = Installation status

AX=0018H: Event handler installed

AX=FFFFH: Event handler could not be installed

Remarks

At least one of bits 5 to 7 must be set in the event mask of the CX register to ensure that the event reacts to at least one of the control keys. If the programmer prefers not to read the Shift, Ctrl or Alt keys along with mouse buttons, use functions 0CH or 14H instead.

An error can occur if three alternate event handlers were previously installed, or if an event handler with the same event mask already exists.

Remarks

The event handler is called by the mouse driver through a FAR call assembler instruction, and therefore must be terminated with a FAR RET instruction. None of the various processor registers may be returned to the caller with a changed content.

The mouse driver passes the following information to the event handler through the processor registers during the call:

AX = Event mask. The bits correspond to the various events as indicated in the CX register during the installation of the event handler. In addition, other bits can be set, since the value reflects the current status of the mouse driver, and is not limited to the selected events.

BX = Mouse button status:

Bit 0 = Left mouse button activated

Bit 1 = Right mouse button activated

Bit 2 = Center mouse button activated

CX = Horizontal mouse position.
 DX = Vertical mouse position.
 SI = Length of last horizontal mouse movement.
 DI = Length of the last vertical mouse movement.
 DS = Data segment of the mouse driver.

The coordinates returned in the CX and DX registers refer to the pixel positions in the virtual mouse display screen rather than physical positions on the actual display screen.

The values in the SI and DI registers refer to mickeys (one mickey = 1/200 inch).

These mickey values must be interpreted as signed numbers. Positive values indicate movement toward the bottom or right border of the screen, while negative values indicate movement toward the top or left border of the screen.

Interrupt 33H, Function 19H	Mouse
Determine address of alternate event handler	

Returns the address of an alternate event handler to the caller.

Input	AX = 19H
	CX = Event handler event mask
Output	CX = 00H: Error
	ES = Segment address of event handler
	DX = Offset address of event handler

Remarks

See the description of function 18H above for additional information about the meanings of each bit in the event mask.

The function call fails if no alternate event handler with the indicated event mask was previously installed.

Interrupt 33H, Function 1AH	Mouse
Set mouse sensitivity	

Defines the relationship between physical mouse movement and mouse cursor movement. Also defines the maximum for doubling mouse speed.

Input	AX = 1AH
	BX = Number of horizontal mickeys
	CX = Number of vertical mickeys
	DX = Maximum limit for doubling the mouse speed
Output	No output

Remarks

Values in the CX and DX registers can range from 1 to 32767.

The default setting is 8 horizontal mickeys and 16 vertical mickeys. This causes the mouse cursor to move twice as fast horizontally as it moves vertically.

To prevent doubling of the mouse speed, the limit can be set higher.

Speeds in excess of 5,000 mickeys per second cannot be achieved by practical means.

Calling function 00H (Reset mouse driver) changes any previously set values to the default values.

Interrupt 33H, Function 1BH	Mouse
Determine mouse sensitivity	

Returns the parameters previously set by calling function 1AH or functions 0FH and 13H.

Input	AX = 1BH
Output	BX = Number of horizontal mickeys
	CX = Number of vertical mickeys
	DX = Maximum limit for doubling the mouse speed

Interrupt 33H, Function 1CH	Mouse
Set mouse hardware interrupt rate	

Determines the frequency at which the mouse hardware reads the current mouse position and mouse button status.

Input	AX = 1CH
	BX = Interrupt rate
	Bit 0: No interrupts
	Bit 1: 30 interrupts per second
	Bit 2: 50 interrupts per second
	Bit 3: 100 interrupts per second
	Bit 4: 200 interrupts per second
	Bits 5-15: Unused
Output	No output

Remarks

This function is only available for the Inport mouse.

If more than one bit is set in the BX register, only the least significant bit which is set counts.

The mouse's resolution increases with the number of interrupts. The increased number of mouse interrupts decreases the speed of the foreground program.

Interrupt 33H, Function 1DH	Mouse
Set display page	

Specifies the display page on which the mouse cursor appears.

Input AX = 1DH
 BX = Number of the display page

Output No output

Remarks

Default value is display page 0.

Calling this function only makes sense if the application program works with several display pages, as available on CGA, EGA and VGA cards.

Interrupt 33H, Function 1EH	Mouse
Determine display page	

Determines the display page on which the mouse cursor appears.

Input AX = 1EH

Output BX = Number of the display page

Interrupt 33H, Function 1FH	Mouse
Disable mouse driver	

Deactivates the current mouse driver and returns the address of the previous interrupt handlers for interrupt 33H.

Input AX = 1FH

Output AX = Error status
 AX=FFFFH: Error
 AX=1FH: O.K.
 ES = Segment address of previous event handler
 BX = Offset address of previous event handler

Remarks

This call releases any previously installed and active mouse driver interrupt routines. The exception to this is the handler for interrupt 33H, but the caller can reload this interrupt vector with its original value since this address is returned in the ES:BX register pair.

Interrupt 33H, Function 20H	Mouse
Enable mouse driver	

Activates a mouse driver previously deactivated by function 1FH.

Input AX = 20H

Output No output

Interrupt 33H, Function 21H	Mouse
Reset mouse driver	

Resets the mouse driver, disables the mouse cursor and disables the currently installed event handler.

Input AX = 21H

Output AX = Error status

AX=FFFFH: Error

AX=0021H: O.K.

BX = Number of mouse buttons

Remarks

Unlike function 00H, this function does not perform a total mouse hardware reset.

Interrupt 33H, Function 22H	Mouse
Set language for messages	

Set language for messages

Specifies language for mouse messages.

Input AX = 22H

BX = Language number

BX=0: English

BX=1: French

BX=2: Dutch

BX=3: German

BX=4: Swedish

BX=5: Finnish

BX=6: Spanish

BX=7: Portuguese

BX=8: Italian

Output No output

Remarks

This function applies only to the mouse driver published for international use. Function 22H is not available on the domestic version of the mouse driver.

Interrupt 33H, Function 23H	Mouse
Get language number	

Returns the number indicating the language under which the mouse driver is operating.

Input AX = 23H

Output BX = Language number

BX=0: English

BX=1: French

BX=2: Dutch

BX=3: German

BX=4: Swedish

BX=5: Finnish

BX=6: Spanish

BX=7: Portuguese

BX=8: Italian

Remarks

This function applies only to the mouse driver published for international use. Function 23H is not available on the domestic version of the mouse driver.

Interrupt 33H, Function 24H	Mouse
Determine mouse type	

Determines the type of mouse installed and the version number of the mouse driver.

Input AX = 24H

Output BH = Whole number of the version number

BL = Fraction of the version number

CH = Mouse type

CH=1: Bus mouse

CH=2: Serial mouse

CH=3: Inport mouse

CH=4: PS/2 mouse

CH=5: HP mouse

CL = IRQ number

CL=0: PS/2

CL=2, 3, 4, 5 or 7: IRQ number in the PC

Remarks

If the version number of the mouse driver is for example 6.24, the value 6 is returned in the BH register and the value 24 is returned in the BL register.

Interrupt 33H, Function 25H	Mouse (Version 6.26 and above)
Get general driver information	

Returns general information describing the mouse driver, such as the driver type and cursor type.

Input AX = 25H

Output AX = General information (see below)

BX = OS/2 status information

CX = OS/2 status information

DX = OS/2 status information

Remarks

The AX register receives the general information as a bit field. These bits have the following meanings:

Bit 15: Type of driver

0(b)=COM file

1(b)=Device driver accessed through CONFIG.SYS

Bit 12 and 13: Mouse cursor information

00(b)=Software text cursor

01(b)=Hardware text cursor

10(b), 11(b)=Graphic cursor

Bits 8 to 11 : Mouse hardware interrupt rate

The arguments returned in the BX, CX and DX registers apply only to mouse drivers in OS/2. They have no significance in DOS programming.

Interrupt 33H, Function 26H	Mouse (Version 6.26 and above)
Get maximum virtual coordinates	

Returns the maximum virtual mouse display coordinates, and indicates whether the mouse driver is active or inactive.

Input	AX = 26H
Output	BX = Mouse driver status BX=0: Inactive BX>0: Active CX = Maximum virtual X-coordinate DX = Maximum virtual Y-coordinate

Remarks

Functions 1FH and 20H control the mouse driver status, returned in the BX register after calling this function.

The values returned in the CX and DX registers describe the size of the virtual mouse display screen, not the cursor positions specified in functions 07H and 08H.

Interrupt 33H, Function 27H	Mouse (Version 7.01 and above)
Get masks and mickey counts	

Reads screen and cursor masks. Also, this function returns information about mouse movement since the last reading.

Input	AX = 27H
Output	AX = AND mask (software cursor) or starting scan line (hardware cursor) BX = XOR mask (software cursor) or ending scan line (hardware cursor) CX = Length of horizontal movement in mickeys DX = Length of vertical movement in mickeys

Remarks

The values returned vary with the type of cursor active during the function call. If the hardware cursor is active, the function receives the starting and ending scan lines of the cursor. If a software cursor is active, the function receives the current screen and cursor mask values.

This function has been supported since Version 7.01 of the mouse driver. Version 7.02 was the first version to return the scan line information describing the hardware cursor.

The movement values returned in the CX and DX registers are taken directly from the mouse hardware, and are not influenced by various software settings such as the threshold value for doubling mouse speed, or the acceleration curve.

Interrupt 33H, Function 28H	Mouse (Version 7.0 and above)
Set video mode	

Sets the video mode if the selected mode is supported by the active video card.

Input

AX = 28H

CX = Video mode number

DX = Screen font size

Output

CX = Function status

Remarks

A list of the available video modes and their code numbers can be queried with function 29H.

The value zero is returned in the CX register if the video mode indicated is supported by the active video-hardware and therefore could be set. Otherwise the code number from the AX register is returned.

The calling parameter in DX is only expected with a few video modes which operate with settable font sizes. In the higher level byte of DX the size of the font, together with the Y-axis and in the lower level byte the extent along the X-axis, must be coded.

Interrupt 33H, Function 29H**Mouse** (Version 7.0 and above)

Count video modes

Gets a numbered list of video modes supported by the active video card.

Input

AX = 29H

CX = Video mode

CX=0: First video mode

CX<>0: Next video mode

Output

BX = Segment of string

CX = Video mode number

DX = Offset of string

Remarks

Multiple calls of this function are required to generate a complete list of supported video modes. The value 0 must be passed in the CX parameter for the first call, with values unequal to 0 passed for any subsequent calls.

Each subsequent call describes a video mode. When the CX register returns 0, this indicates that all available video modes have been read.

Function 29H doesn't directly return the video mode's type (text or graphic), resolution or color capability. This information can usually be obtained from an ASCII string whose address is returned in the BX:DX register pair. If this ASCII string is available, and if the BX:DX registers contain values other than 0, the ASCII string ends with a dollar sign and a null byte.

Interrupt 33H, Function 2AH**Mouse** (Version 7.02 and above)

Get cursor hotspot

Returns information about the mouse type and cursor hotspot.

Input	AX = 2AH
Output	AX = Internal cursor flag
	BX = Hotspot X-coordinate
	CX = Hotspot Y-coordinate
	DX = Type of mouse

Remarks

The internal cursor flag indicates whether the mouse cursor is visible or not. Functions 01H and 02H indirectly influence this flag. A value of 0 signals that the mouse cursor is currently invisible. Any other value indicates that the mouse cursor is currently visible.

The hotspot is the pixel in a graphic cursor mask whose position is returned when reading cursor position. Its distance from the upper-left corner of the bit mask is returned to the BX and CX registers as a signed integer. This integer can range from -128 to 127).

The DX register indicates the mouse type. This type code can be one of the following:

- 0 = No mouse
- 1 = Bus mouse
- 2 = Serial mouse
- 3 = InPort mouse
- 4 = PS/2 mouse
- 5 = Hewlett-Packard mouse

Interrupt 33H, Function 2BH**Mouse** (*Version 7.0 and above*)

Set acceleration curves

All four acceleration curves, which the mouse-driver administers internally, can be loaded with the help of this function and one can be selected as the current one.

Input	AX = 2BH
	BX = Number of curve to activate
	ES = Segment address of curve array
	SI = Offset address of curve array

Output AX = FFFFH: Error

AX = 0000H: O.K.

Remarks

This function changes the preset acceleration curves in the driver, by passing the value -1 as the number of the current acceleration curve. The passing of a buffer with the data of the acceleration curves is not required in this case.

The four acceleration curves are described through a data structure which is created by the caller in memory and must be passed using the ES:SI registers. See Chapter XXX for more information about this structure.

Interrupt 33H, Function 2CH	Mouse (Version 7.0 and above)
Read acceleration curves	

Reads the acceleration curves.

Input	AX = 2CH
Output	AX = 00H: O.K. BX=Number of the current acceleration curve (0 to 3) ES=Segment address of curve array SI=Offset address of curve array AX = FFFH: Error

Remarks

The AX register should be read following every call of this function. The acceleration curves could only be read if the AX register contained a value of 0.

The contents of the ES:SI register pair indicate the buffer containing the array describing the current acceleration curve. This data structure corresponds to the format used in function 2BH for setting an acceleration curve.

Interrupt 33H, Function 2DH	Mouse (Version 7.0 and above)
Get/set active acceleration curves	

Activates one of the four acceleration curves, and reads the current acceleration curve.

Input	AX = 2DH BX = -1: Get current acceleration curve BX = 1 - 4: Set current acceleration curve
Output	AX = 0: O.K. BX=Number of current active acceleration curve ES=Segment address of ASCII string describing acceleration curve SI=Offset address of ASCII string describing acceleration curve AX=-2: Bad curve number

Remarks

If the value -1 is passed in the BX register as part of the function call, this function returns information describing the current acceleration curve in the BX, ES and SI registers. If a value from 1 to 4 is passed in the BX register, the corresponding acceleration curve becomes active. If this is the case, the BX register returns the current acceleration curve number.

After setting the current acceleration curve, the ES:SI registers indicate the ASCII string containing acceleration curve data (see function 2BH). This string contains 16 bytes, and has no special end character (i.e., null byte or \$). This string provides the symbolic name of the acceleration curve.

Interrupt 33H, Function 2EH	Mouse (<i>Version 1.0 and above</i>)
Undocumented	

This undocumented function is used exclusively by the mouse driver, and cannot be called from a program.

Interrupt 33H, Function 2FH	Mouse (<i>Version 7.02 and above</i>)
Mouse hardware reset	

Resets the mouse hardware without affecting the software configuration (mouse cursor appearance, threshold value, acceleration curve settings, etc.).

Input AX = 2FH

Output AX = FFFFH: O.K.

 AX = 0: Error

Remarks

This function is the hardware equivalent of function 21H, which resets the software parameters specified in the mouse driver.

Interrupt 33H, Function 30H	Mouse (<i>Version 7.04 and above</i>)
Get/set ballpoint information	

This function has been tailored specifically for the needs of the ballpoint mouse.

Input AX = 30H

 BX = Angle of rotation

 CX = Command code

Output AX = Function status

 BX = Angle of rotation

 CX = Active buttons

Remarks

After the function call, the function status in AX should be checked immediately, because the value -1 indicates that no Ballpoint-Mouse is installed. Any other value indicates however the existence of a Ballpoint-Mouse and reflects the status of the various mouse-buttons. The individual buttons are represented by the following bits in the AX register:

Bit 2 = Button 4

Bit 3 = Button 2

Bit 4 = Button 3

Bit 5 = Button 1

The remaining bits contain the value 0.

If the current angle of rotation and the active buttons are queried with this function, the CX register must be loaded with the value zero before the function call. An angle of rotation in BX is not required at this time.

As a function result, the angle of rotation is returned in the BX register. It is a value between 0 and 360 (degrees). The two active buttons can be read from the high byte of CX, while the inactive buttons are coded into the low byte of this register. In these two bytes are the following bits for the individual buttons:

Bit 2 = Button 4

Bit 3 = Button 2

Bit 4 = Button 3

Bit 5 = Button 1

the remaining bits contain the value 0.

If the mouse-driver should be informed with the help of this function of the current angle of rotation of the Ballpoint-Mouse and the two active buttons selected, the active and inactive buttons must be coded into the CX register. The coding is exactly as in the return for the active and inactive buttons after a query (see above). In addition, a value between 0 and 360 degrees is expected as the angle of rotation in BX.

Interrupt 33H, Function 31H	Mouse (Version 7.05 and above)
Get minimum/maximum virtual coordinates	

Returns current minimum and maximum coordinates of the virtual mouse display screen in the current video mode.

Input

AX = 31H

Output

AX = Minimum X-coordinate

BX = Minimum Y-coordinate

CX = Maximum X-coordinate

DX = Maximum Y-coordinate

Remarks

Functions 07H and 08H affect the size of the virtual mouse display screen.

Interrupt 33H, Function 32H	Mouse (Version 7.05 and above)
Get active advanced functions	

Returns information describing advanced functions supported by the mouse driver, and not accessible from function 25H.

Input

AX = 32H

Output

AX = Supported functions

Remarks

The function result in AX is a bit field, where each bit stands for a function. If it is set, the function is supported.

Remarks

The AX register receives the function support as a bit field. These bits have the following meanings:

Bit 15=Function 25H

Bit 14=Function 26H

Bit 13=Function 27H

Bit 12=Function 28H

Bit 11=Function 29H

Bit 10=Function 2AH

Bit 9=Function 2BH

Bit 8=Function 2CH

Bit 7=Function 2DH

Bit 6=Function 2EH

Bit 5=Function 2FH

Bit 4=Function 30H

Bit 3=Function 31H

Bit 1=Function 32H

Bit 0=Function 33H

Interrupt 33H, Function 33H**Mouse** (*Version 7.05 and above*)

Get switch settings

Returns all mouse parameters in the mouse driver that can be set through hardware or software.

Input

AX = 33H

CX = Buffer length

ES = Segment address of buffer

DX = Offset address of buffer

Output

AX = 0

CX = Number of bytes in buffer

ES = Segment address of buffer

DX = Offset address of buffer

Remarks

The buffer has the following structure:

Offset	Content	Range
0	Mouse type (low nibble)	0-5
0	Mouse type (high nibble)	0-4
1	Language	0-10
2	Horizontal sensitivity	0-100
3	Vertical sensitivity	0-100
4	Double threshold	0-100
5	Ballistic curve	1-4
6	Interrupt rate	1-4
7	Cursor override mask	0-255
8	Laptop adjustment	0-255
9	Memory type	0-2
10	Super VGA support	0-1
11	Rotation angle	0-359
13	Primary button	1-4
14	Secondary button	1-4
15	Click lock enabled	0-1
16	Acceleration curve data	Bytes 16-339

Interrupt 33H, Function 34H**Mouse** (*Version 8.0 and above*)

Get MOUSE.INI location

Returns the exact path designation of the MOUSE.INI file as an ASCII string. This function applies only to Microsoft Windows.

Input AX = 34H

Output AX = 0

ES = Segment address of buffer

DX = Offset address of buffer

Remarks

The ASCII string is terminated by a null byte.

The path of MOUSE.INI is obtained from the MOUSE variable. If this variable was not defined, MOUSE.INI is assumed to be in the directory containing a mouse driver.

Hardware Interrupts

Interrupt 00H	Hardware (CPU)
Division by zero	

The CPU calls this interrupt when it encounters a divisor of 0 during one of the two assembly language division instructions (DIV or IDIV). According to the rules of mathematics, dividing a number by 0 is illegal. During the booting process, this interrupt points to a routine that, when called, displays the "Division by Zero" error message (or a similar message) on the screen. The interrupt continues with the execution of the current program.

Interrupt 01H	Hardware (CPU)
Single step	

The CPU calls this interrupt when the TRAP bit in the flag register of the CPU has been set to 1. Then the interrupt is called after the execution of each assembly language instruction. This allows the user to follow these instructions, determine the changes in register contents and determine which instructions are executed. To prevent the call of the interrupt after the execution of every instruction in the trap routine (which would create an endless loop and a stack overflow), the processor resets the TRAP bit upon entry to the trap routine. If the trap routine ends with the IRET instruction, it automatically resets the TRAP bit to its old value by restoring the complete flag register from the stack. Because of this, the execution of the next instruction calls interrupt 1 again. Once the programmer has obtained the necessary information about a program from single step mode, the TRAP mode (or TRAP bit) can be disabled.

Interrupt 02H	Hardware (CPU)
NMI	

The hardware calls this interrupt when an error is discovered in the RAM chips. The system calls the non-maskable interrupt because this type of error impairs the capabilities of the system, and can lead to a crash. The NMI has the highest priority of all interrupts and therefore is executed faster than other interrupts. The NMI usually calls a BIOS routine which informs the user of a memory error, lists the number of defective memory chips and stops the system.

If the NMI detects an error, the math coprocessor included in some PCs can also trigger the NMI. Even though NMI usually cannot be suppressed, the PC allows an exception to this rule. Some PC/XT and AT models have a special port (port A0H on PCs and XTs, port 70H on ATs). If a 0 value is written to one of these ports, the NMI interrupt is disabled. If the ports return the value 80H, the NMI interrupt is enabled.

Interrupt 03H	Hardware (CPU)
Breakpoint	

While the other interrupts can be called with a two-byte assembly language instruction (first byte CDH, second byte the number of the interrupt), interrupt 3 is called by the single-byte instruction CCH. This interrupt can be used to test programs when you want to execute the program up to a certain instruction, then stop and display the current register contents. Utilities

designed for program testing like DEBUG implement this by placing calls for interrupt 3 where the break should occur. When the program is executed and the processor reaches the instruction, it calls interrupt 3. The program testing utility contains a routine which displays the register contents and other information.

Interrupt 04H	Hardware (CPU)
Overflow	

This interrupt can be called by the INTO (INTerrupt on Overflow) conditional assembly language instruction. The call occurs when the overflow bit in the flag register is set during the execution of the INTO instruction. This can happen following math operations (e.g., multiplication with the MUL instruction) that produce a result which cannot be represented within a specified number of bits. This interrupt can also be called with the normal INT instruction, but this instruction isn't controlled by the status of the set overflow bit. Since it is seldom used, DOS points this interrupt to an IRET instruction.

Interrupt 05H	BIOS
Hardcopy	

BIOS calls this interrupt when the user presses the **Prt Sc** key. The system then makes a hardcopy by sending the current screen contents to a printer. BIOS initializes the interrupt vector from the vector table and points to the BIOS hardcopy routine in ROM-BIOS. Assembly language and programs written in higher .High level languages can use this interrupt with the INT instruction to get a hardcopy during program execution.

Interrupt 08H	Hardware (8259 interrupt controller)
Timer	

In the PC, the 8259 timer chip receives 1,193,180 signals per second from the heart of the system, which is an oscillating quartz crystal. After 65,536 of these signals (1 second), it triggers a call of interrupt 8, which the 8259 transmits to the CPU. Since the frequency of the call of this interrupt is independent of the system clock frequency, interrupt 8 works well for timekeeping. The PC also uses the interrupt for timekeeping. BIOS points the interrupt vector of this interrupt to its own routine, which is called 18.2 times per second. A time counter increments every second and disables the disk drive motor if disk access hasn't occurred within a certain time period.

Interrupt 09H	Hardware (8259 interrupt controller)
Keyboard	

PC keyboards contain an independent processor. This Intel processor carries either the number 8048 (PC/XT) or 8042 (AT). This processor monitors the keyboard and registers whether a key was depressed or released. When either of these actions occur, this processor must inform the CPU so the code of the activated key can be sent to the system and processed. The keyboard instructs the interrupt controller to call interrupt 9. This interrupt calls a BIOS routine that reads the character from the keyboard and places it into the keyboard buffer.

Introduction To Number Systems

We've often mentioned numbers in the binary system and hexadecimal systems instead of the normal decimal system. This Appendix presents a brief introduction to these number systems.

Decimal system

Before explaining the new number systems, you should know the basic concepts of the decimal system. The decimal number 1989 can also be written as $1*1000+9*100+8*10+9*1$. This shows that if you number the digits from right to left, the first number represents a column of ones, the second number represents a column of tens, the third number represents a column of hundreds and the fourth number represents a column of thousands. The numbers increase from right to left in powers of 10.

The first digit of any number system has the value 1. The factor by which the value increases from one column to the next differs among the number systems. This factor corresponds to the numbers with which the number system works. The factor is 10 with the decimal system because ten different numbers are available for each digit (0 to 9).

This principle of powers for each column also applies to the binary and hexadecimal systems.

Binary system

Since a computer recognizes the numbers 0 and 1 on its lowest functional level, the binary system is essential to computing. The value of the numbers double from column to column because the binary system only uses powers of two for each column (i.e., the numbers 0 and 1 instead of the numbers 0 to 9).

Now let's count the binary places starting from right to left as we did in the decimal example described above. The first (right hand) position counts as one, the second as two, the third as four and the fourth as eight. The places then follow as 16, 32, 64, 128, etc.

For example, 11001 binary converts to 25 decimal, or the equation $1*16+1*8+0*4+0*2+1*1$.

Hexadecimal system

Unlike the binary system, the hexadecimal system; operates with more basic numbers than the decimal system. This system counts single digits from 0 to F. Since only the ten numbers of the decimal system are able to represent a number, the numbers from 10 to 15 in hexadecimal use the letters A to F in addition to the numbers 0 to 9. AH represents 10, BH for 11, CH for 12, DH for 13, EH for 14 and FH for 15.

By using 16 numbers or letters for each position, the value by which each position increments is 16.

The first position has the value 1, the second 16, the third 256 and the fourth 4,096.

For example, the hexadecimal number FB3H converts into 4,019 decimal, or $15*256+11*16+3*1$.

Hex and binary

The hexadecimal system and the binary system are easily converted back and forth. For example, one four-digit binary number converts to a single-digit hexadecimal number. Because of this, the hexadecimal system is an important part of assembly language programming. It's much simpler to convey a byte (an eight-bit number) using two hexadecimal digits than it is for the developer to compute a 16-bit binary equivalent.

This book denotes all binary numbers by the letter b, and all hexadecimal numbers by the letter H.

The following tables should help explain number systems more clearly.

Number of positions in each number system

Positions	Decimal	Binary	Hexadecimal
1	10000	1000	100
2	16	8	4
4	65536	4096	256
8			16
16			1

Comparing selected numbers in each number system

Decimal	Binary	Hexadecimal
0	0(b)	0H
1	1(b)	1H
2	10(b)	2H
3	11(b)	3H
4	100(b)	4H
5	101(b)	5H
6	110(b)	6H
7	111(b)	7H
8	1000(b)	8H
9	1001(b)	9H
10	1010(b)	AH
11	1011(b)	BH
12	1100(b)	CH
128	10000000(b)	80H
129	10000001(b)	81H
256	100000000(b)	100H
1024	10000000000(b)	400H
4096	1000000000000(b)	1000H
65535	1111111111111111(b)	FFFFH

Program Listings From The PC INTERN CD-ROM

ASM

BASIC

WINDOWS 95

C

PASCAL

ASM Listings

ATCLK.ASM
CEBHAND.ASM
CONDRV.ASM
DUMPA.ASM
EXEC.ASM
EXESYS.ASM

FF.ASM
GETSCAN.ASM
MACROKEY.ASM
PRCVT.ASM
RAMDISK.ASM
SOUNDA.ASM

TESTCOM.ASM
TESTEXE.ASM
VCOL.ASM
VHERC.ASM
VMONO.ASM

BASIC Listings

9HDEMO.BAS
CONFIGB.BAS
DIRB.BAS
DVIB.BAS
FIXPARTB.BAS

JOYSTB.BAS
KEYB.BAS
LEDB.BAS
MCBB.BAS
MEDIAIDB.BAS

MF2B.BAS
NOKEYB.BAS
RTCB.BAS
WINDAB.BAS

C Program Listings

9HDEMOBC.C
9HDEMOMC.C
ARGS.C
ARGS.H
CDDEV.C
CDDEV.H
CDROM.C
CDUTIL.C
CDUTIL.H
CONFIGC.C
CTTS.C
CTTS.H
DBLSPCC.C
DDPTC.C
DFC.C
DIRC1.C
DIRC2.C
DMAUTIL.C
DMAUTIL.H
DSP.C
DSPUTIL.C
DSPUTIL.H
DUMPC.C
DVIC.C
EDIT.C
EDIT.H
EMMC.C
EXTC.C
FIXPARTC.C
FLOCKC.C
FM.C
FMUTIL.C
FMUTIL.H
HMAC.C
HMACA.ASM
IRQSTAT.C

IRQUTIL.C
IRQUTIL.H
ISEVC.C
JOYSTC.C
KBD.H
KEYC.C
LEDC.C
LOGOC.C
LOGOCA.ASM
MCBC.C
MEDIAIDC.C
MEMDEMOC.C
MF2C.C
MIKADOC.C
MIX.C
MIXUTIL.C
MIXUTIL.H
MOUSEC.C
MOUSECA.ASM
MRCIC.C
NETFILEC.C
NOKEYC.C
PLINKC.C
PLINKCA.ASM
PROCC.C
PROCCA.ASM
RAWCOOK.C
RECLOCKC.C
RTCC.C
S3220C.C
S3220CA.ASM
S3240C.C
S3240CA.ASM
S6435C.C
S6435CA.ASM
SBUTIL.C

SBUTIL.H
SERIRQ.C
SERTRANS.C
SERTRANS.H
SERUTIL.C
SERUTIL.H
SOFTSCCA.ASM
SOFTSCRC.C
SOUNDC.C
SOUNDCA.ASM
TSRC.C
TSRCA.ASM
TYPES.H
TYPMC.C
TYPMCA.ASM
V16COLC.C
V16COLCA.ASM
V3220C.C
V3220CA.ASM
V3240C.C
V3240CA.ASM
V8060C.C
V8060CA.ASM
VDACC.C
VIDEOC.C
VIOSC.C
VIOSCA.ASM
VOICE.C
VONOFFC.C
WIN.C
WIN.H
WINDAC.C
XMSC.C
XMSCA.ASM

PASCAL Program Listings

9HDEMOP.PAS
ARGS.PAS
ASZDEMO.PAS
CDDEV.PAS
CDROM.PAS
CDUTIL.PAS
CONFIGP.PAS
CTTS.PAS
DBLSPCP.PAS
DDPTP.PAS
DFP.PAS
DIRP1.PAS
DIRP2.PAS
DMAUTIL.PAS
DSP.PAS
DSPUTIL.PAS
DUMPP.PAS
DVIP.PAS
EMMP.PAS
EXTP.PAS
FIXPARTP.PAS
FLOCKP.PAS
FM.PAS
FMUTIL.PAS
HMAP.PAS
IRQUTIL.PAS
ISEVP.PAS
JOYSTP.PAS

KEYP.PAS
LEDP.PAS
LOGOP.PAS
MCBP.PAS
MEDIAIDP.PAS
MEMDEMOP.PAS
MF2P.PAS
MIKADOP.PAS
MIX.PAS
MIXUTIL.PAS
MOUSEP.PAS
MOUSEPA.ASM
MRCIP.PAS
NETFILEP.PAS
NOKEYP.PAS
PLINKP.PAS
PLINKPA.ASM
PROCP.PAS
PROCPA.ASM
RAWCOOKP.PAS
RECLOCKP.PAS
RTCP.PAS
S3220P.PAS
S3220PA.ASM
S3240P.PAS
S3240PA.ASM
S6435P.PAS
S6435PA.ASM

SBUTIL.PAS
SERIRQ.PAS
SERTRANS.PAS
SERUTIL.PAS
SOFTSCR.PAS
TSRP.PAS
TSRPA.ASM
TYPMP.PAS
TYPMPA.ASM
V16COLP.PAS
V16COLPA.ASM
V3220P.PAS
V3220PA.ASM
V3240P.PAS
V3240PA.ASM
V8060P.PAS
V8060PA.ASM
VDACP.PAS
VIDEOP.PAS
VIOSP.PAS
VIOSPA.ASM
VOICE.PAS
VONOFFP.PAS
WIN.PAS
WINDAP.PAS
XMSP.PAS

WINDOWS 95 Program Listings

BADBOYS

BADBOYS.C

BROWSE

BROWSE.C
PAINTPIC.C
PAINTPIC.H

CHATTER

CHATTER.C
CTL3D.H
ISDNFS.H
ISDNPRO.H

CONTROL

CONTROL.C
ENUMFOLD.C
ENUMFOLD.H

DD

DD.C
ENUMFOLD.C
ENUMFOLD.H

DESKTOP

DESKTOP.C
SHORTCUT.C
SHORTCUT.H

DINNER

DINNER.C
TURTLE32.C
TURTLE32.H

DRAGDROP

DRAGDROP.C

ENUMFOLD

ENUMFOLD.C
ENUMFOLD.H

FINDEXEC

FINDEXEC.C

GETEXE

GETEXE.C

GETICONS

GETICONS.C

IMAGEDIT

IMAGEDIT.C
PAINTPIC.H

LVDEMO

LVDEMO.C
PAINTPIC.C
PAINTPIC.H

MASTER32

MASTER32.C
PRIVAT.H
PRIVAT.H

MEMSTAT

MEMSTAT.C

MEMWAL

MEMWAL.BAS
TOOLHE1.BAS
TOOLHE2.BAS

MEMDUMP

MEMDUMP.C
MULTIPLE.C
TURTLE.H

MULTIPLE

MULTIPLE.BAS

OPENSA

OPENSA.C

PAINTPIC

PAINTPIC.C
PAINTPIC.H

PEIMEX

PEIMEX.C

PERSIST

PERSIST.C

PRINTER

ENUMFOLD.C
ENUMFOLD.H
PRINTER.C
TASKBAR.C
TASKBAR.H

PROGRAM

ENUMFOLD.C
ENUMFOLD.H
PAINTPIC.C
PAINTPIC.H
PROGRAM.C
SHORTCUT.C
SHORTCUT.H

PYTHAGO

PYTHAGO.C
TURTLE32.C
TURTLE32.H

REGSTAT

REGSTAT.C

SHORTCUT

SHORTCUT.C
SHORTCUT.H

SLAVE32

SLAVE32.C
TURTLE32.C
TURTLE32.H

SHFILE

SHFILE.C

TURTLE

TURTLE.C
TURTLE.H

TURTLE32

TURTLE32.C
TURTLE32.H

TVDEMO

PAINTPIC.C
PAINTPIC.H
TVDEMO.C

URL

URL.C

VIRALLOC

VIRALLOC.C