MS-DOS, PC-BIOS, and File I/O — Chapter 13

A typical PC system consists of many component besides the 80x86 CPU and memory. MS-DOS and the PC's BIOS provide a software connection between your application program and the underlying hardware. Although it is sometimes necessary to program the hardware directly yourself, more often than not it's best to let the system software (MS-DOS and the BIOS) handle this for you. Furthermore, it's much easier for you to simply call a routine built into your system than to write the routine yourself.

You can access the IBM PC system hardware at one of three general levels from assembly language. You can program the hardware directly, you can use ROM BIOS routines to access the hardware for you, or you can make MS-DOS calls to access the hardware. Each level of system access has its own set of advantages and disadvantages.

Programming the hardware directly offers two advantages over the other schemes: control and efficiency. If you're controlling the hardware modes, you can get that last drop of performance out of the system by taking advantage of special hardware tricks or other details which a general purpose routine cannot. For some programs, like screen editors (which must have high speed access to the video display), accessing the hardware directly is the only way to achieve reasonable performance levels.

On the other hand, programming the hardware directly has its drawbacks as well. The screen editor which directly accesses video memory may not work if a new type of video display card appears for the IBM PC. Multiple display drivers may be necessary for such a program, increasing the amount of work to create and maintain the program. Furthermore, had you written several programs which access the screen memory directly and IBM produced a new, incompatible, display adapter, you'd have to rewrite all your programs to work with the new display card.

Your work load would be reduced tremendously if IBM supplied, in a fixed, known, location, some routines which did all the screen I/O operations for you. Your programs would all call these routines. When a manufacturer introduces a new display adapter, it supplies a new set of video display routines with the adapter card. These new routines would patch into the old ones (replacing or augmenting them) so that calls to the old routines would now call the new routines. If the program interface is the same between the two set of routines, your programs will still work with the new routines.

IBM has implemented such a mechanism in the PC system firmware. Up at the high end of the one megabyte memory space in the PC are some addresses dedicated to ROM data storage. These ROM memory chips contain special software called the PC Basic Input Output System, or BIOS. The BIOS routines provide a hardware-independent interface to various devices in the IBM PC system. For example, one of the BIOS services is a video display driver. By making various calls to the BIOS video routines, your software will be able to write characters to the screen regardless of the actual display board installed.

At one level up is MS-DOS. While the BIOS allows you to manipulate devices in a very low level fashion, MS-DOS provides a high-level interface to many devices. For example, one of the BIOS routines allows you to access the floppy disk drive. With this BIOS routine you may read or write blocks on the diskette. Unfortunately, the BIOS doesn't know about things like files and directories. It only knows about blocks. If you want to access a file on the disk drive using a BIOS call, you'll have to know exactly where that file appears on the diskette surface. On the other hand, calls to MS-DOS allow you to deal with filenames rather than file disk addresses. MS-DOS keeps track of where files are on the disk surface and makes calls to the ROM BIOS to read the appropriate blocks for you. This high-level interface greatly reduces the amount of effort your software need expend in order to access data on the disk drive.

The purpose of this chapter is to provide a brief introduction to the various BIOS and DOS services available to you. This chapter does not attempt to begin to describe all of the routines or the options available to each routine. There are several other texts the size of this one which attempt to discuss *just* the BIOS or *just* MS-DOS. Furthermore, any attempt to provide complete coverage of MS-DOS or the BIOS in a single text is doomed to failure from the start– both are a moving target with specifications changing with each new version. So rather than try to explain everything, this chapter will simply attempt to present the flavor. Check in the bibliography for texts dealing directly with BIOS or MS -DOS.

13.0 Chapter Overview

This chapter presents material that is specific to the PC. This information on the PC's BIOS and MS-DOS is not necessary if you want to learn about assembly language programming; however, this is important information for anyone wanting to write assembly language programs that run under MS-DOS on a PC compatible machine. As a result, most of the information in this chapter is optional for those wanting to learn generic 80x86 assembly language programming. On the other hand, this information is handy for those who want to write applications in assembly language on a PC.

The sections below that have a "•" prefix are essential. Those sections with a "❏" discuss advanced topics that you may want to put off for a while.

- • The IBM PC BIOS
- ❏ Print screen.
- • Video services.
- ❏ Equipment installed.
- ❏ Memory available.
- ❏ Low level disk services
- • Serial I/O.
- ❏ Miscellaneous services.
- • Keyboard services.
- • Printer services.
- ❏ Run BASIC.
- ❏ Reboot computer.
- ❏ Real time clock.
- • MS-DOS calling sequence.
- • MS-DOS character functions
- ❏ MS-DOS drive commands.
- ❏ MS-DOS date and time functions.
- ❏ MS-DOS memory management functions.
- ❏ MS-DOS process control functions.
- • MS_DOS "new" filing calls.
- • Open file.
- • Create file.
- • Close file.
- • Read from a file.
- • Write to a file.
- ❏ Seek.
- ❏ Set disk transfer address.
- ❏ Find first file.
- ❏ Find next file.
- • Delete file.
- • Rename file.
- ❏ Change/get file attributes.
- ❏ Get/set file date and time.
- ❏ Other DOS calls
- • File I/O examples.
- • Blocked file I/O.
- ❏ The program segment prefix.
- ❏ Accessing command line parameters.
- ❏ ARGC and ARGV.
- • UCR Standard Library file I/O routines.

- FOPEN.
- FCREATE.
- FCLOSE.
- FFLUSH.
- FGETC.
- FREAD.
- FPUTC
- FWRITE.

❏ Redirection I/O through the STDLIB file I/O routines.

13.1 The IBM PC BIOS

Rather than place the BIOS routines at fixed memory locations in ROM, IBM used a much more flexible approach in the BIOS design. To call a BIOS routine, you use one of the 80x86's int software interrupt instructions. The int instruction uses the following syntax:

int *value*

Value is some number in the range 0..255. Execution of the int instruction will cause the 80x86 to transfer control to one of 256 different interrupt handlers. The interrupt vector table, starting at physical memory location 0:0, holds the addresses of these interrupt handlers. Each address is a full segmented address, requiring four bytes, so there are 400h bytes in the interrupt vector table -- one segmented address for each of the 256 possible software interrupts. For example, int 0 transfers control to the routine whose address is at location 0:0, int 1 transfers control to the routine whose address is at 0:4, int 2 via 0:8, int 3 via 0:C, and int 4 via 0:10.

When the PC resets, one of the first operations it does is initialize several of these interrupt vectors so they point at BIOS service routines. Later, when you execute an appropriate int instruction, control transfers to the appropriate BIOS code.

If all you're doing is calling BIOS routines (as opposed to writing them), you can view the int instruction as nothing more than a special call instruction.

13.2 An Introduction to the BIOS' Services

The IBM PC BIOS uses software interrupts 5 and 10h..1Ah to accomplish various operations. Therefore, the int 5, and int 10h.. int 1ah instructions provide the interface to BIOS. The following table summarizes the BIOS services:

INT	Function
5	Print Screen operation.
10h	Video display services.
11h	Equipment determination.
12h	Memory size determination.
13h	Diskette and hard disk services.
14h	Serial I/O services.
15h	Miscellaneous services.
16h	Keyboard services.
17h	Printer services.
18h	BASIC.
19h	Reboot.
1Ah	Real time clock services.

Most of these routines require various parameters in the 80x86's registers. Some require additional parameters in certain memory locations. The following sections describe the exact operation of many of the BIOS routine.

13.2.1 INT 5- Print Screen

Instruction: int 5h
BIOS Operation: Print the current text screen.
Parameters: None

If you execute the int 5h instruction, the PC will send a copy of the screen image to the printer exactly as though you'd pressed the PrtSc key on the keyboard. In fact, the BIOS issues an int 5 instruction when you press the PrtSc, so the two operations are absolutely identical (other than one is under software control rather than manual control). Note that the 80286 and later also uses int 5 for the BOUNDS trap.

13.2.2 INT 10h - Video Services

Instruction: int 10h
BIOS Operation: Video I/O Services
Parameters: Several, passed in ax, bx, cx, dx, and es:bp registers.

The int 10h instruction does several video display related functions. You can use it to initialize the video display, set the cursor size and position, read the cursor position, manipulate a light pen, read or write the current display page, scroll the data in the screen up or down, read and write characters, read and write pixels in a graphics display mode, and write strings to the display. You select the particular function to execute by passing a value in the ah register.

The video services represent one of the largest set of BIOS calls available. There are many different video display cards manufactured for PCs, each with minor variations and often each having its own set of unique BIOS functions. The BIOS reference in the appendices lists some of the more common functions available, but as pointed out earlier, this list is quite incomplete and out of date given the rapid change in technology.

Probably the most commonly used video service call is the character output routine:

Name: Write char to screen in TTY mode
Parameters ah = 0Eh, al = ASCII code (In graphics mode, bl = Page number)

This routine writes a single character to the display. MS-DOS calls this routine to display characters on the screen. The UCR Standard Library also provides a call which lets you write characters directly to the display using BIOS calls.

Most BIOS video display routines are poorly written. There is not much else that can be said about them. They are extremely slow and don't provide much in the way of functionality. For this reason, most programmers (who need a high-performance video display driver) end up writing their own display code. This provides speed at the expense of portability. Unfortunately, there is rarely any other choice. If you need functionality rather than speed, you should consider using the ANSI.SYS screen driver provided with MS-DOS. This display driver provides all kinds of useful services such as clear to end of line, clear to end of screen, etc. For more information, consult your DOS manual.

Table 49: BIOS Video Functions (Partial List)

AH	Input Parameters	Output Parameters	Description
0	al=mode		Sets the video display mode.
1	ch- Starting line. cl- ending line		Sets the shape of the cursor. Line values are in the range 0..15. You can make the cursor disappear by loading ch with 20h.

Table 49: BIOS Video Functions (Partial List)

AH	Input Parameters	Output Parameters	Description
2	bh- page dh- y coordinate dl- x coordinate		Position cursor to location (x,y) on the screen. Generally you would specify page zero. BIOS maintains a separate cursor for each page.
3	bh- page	ch- starting line cl- ending line dl- x coordinate dh- y coordinate	Get cursor position and shape.
4			Obsolete (Get Light Pen Position).
5	al- display page		Set display page. Switches the text display page to the specified page number. Page zero is the standard text page. Most color adapters support up to eight text pages (0..7).
6	al- Number of lines to scroll. bh- Screen attribute for cleared area. cl- x coordinate UL ch- y coordinate UL dl- x coordinate LR dh- y coordinate LR		Clear or scroll up. If al contains zero, this function clears the rectangular portion of the screen specified by cl/ch (the upper left hand corner) and dl/dh (the lower right hand corner). If al contains any other value, this service will scroll that rectangular window up the number of lines specified in al.
7	al- Number of lines to scroll. bh- Screen attribute for cleared area. cl- x coordinate UL ch- y coordinate UL dl- x coordinate LR dh- y coordinate LR		Clear or scroll down. If al contains zero, this function clears the rectangular portion of the screen specified by cl/ch (the upper left hand corner) and dl/dh (the lower right hand corner). If al contains any other value, this service will scroll that rectangular window down the number of lines specified in al.
8	bh- display page	al- char read ah- char attribute	Read character's ASCII code and attribute byte from current screen position.
9	al- character bh- page bl- attribute cx- # of times to replicate character		This call writes cx copies of the character and attribute in al/bl starting at the current cursor position on the screen. It does not change the cursor's position.
0Ah	al- character bh- page		Writes character in al to the current screen position using the existing attribute. Does not change cursor position.
0Bh	bh- 0 bl- color		Sets the border color for the text display.
0Eh	al- character bh- page		Write a character to the screen. Uses existing attribute and repositions cursor after write.
0Fh		ah- # columns al- display mode bh- page	Get video mode

Note that there are many other BIOS 10h subfunctions. Mostly, these other functions deal with graphics modes (the BIOS is too slow for manipulating graphics, so you shouldn't use those calls) and extended features for certain video display cards. For more information on these calls, pick up a text on the PC's BIOS.

13.2.3 INT 11h - Equipment Installed

Instruction: int 11h
BIOS Operation: Return an equipment list
Parameters: On entry: None, on exit: AX contains equipment list

On return from int 11h, the AX register contains a bit-encoded equipment list with the following values:

Bit 0	Floppy disk drive installed
Bit 1	Math coprocessor installed
Bits 2,3	System board RAM installed (obsolete)
Bits 4,5	Initial video mode 00- none 01- 40x25 color 10- 80x25 color 11- 80x25 b/w
Bits 6,7	Number of disk drives
Bit 8	DMA present
Bits 9,10,11	Number of RS-232 serial cards installed
Bit 12	Game I/O card installed
Bit 13	Serial printer attached
Bits 14,15	Number of printers attached.

Note that this BIOS service was designed around the original IBM PC with its very limited hardware expansion capabilities. The bits returned by this call are almost meaningless today.

13.2.4 INT 12h - Memory Available

Instruction: int 12h
BIOS Operation: Determine memory size
Parameters: Memory size returned in AX

Back in the days when IBM PCs came with up to 64K memory installed on the motherboard, this call had some meaning. However, PCs today can handle up to 64 *megabytes* or more. Obviously this BIOS call is a little out of date. Some PCs use this call for different purposes, but you cannot rely on such calls working on any machine.

13.2.5 INT 13h - Low Level Disk Services

Instruction: int 13h
BIOS Operation: Diskette Services
Parameters: ax, es:bx, cx, dx (see below)

The int 13h function provides several different low-level disk services to PC programs: Reset the diskette system, get the diskette status, read diskette sectors, write diskette sectors, verify diskette sectors, and format a diskette track and many more. This is another example of a BIOS routine which has changed over the years. When this routine was first developed, a 10 megabyte hard disk was considered large. Today, a typical high performance game requires 20 to 30 megabytes of storage.

Table 50: Some Common Disk Subsystem BIOS Calls

AH	Input Parameters	Output Parameters	Description
0	dl- drive (0..7fh is floppy, 80h..ffh is hard)	ah- status (0 and carry clear if no error, error code if error).	Resets the specified disk drive. Resetting a hard disk also resets the floppy drives.
1	dl- drive (as above)	ah- 0 al- status of previous disk operation.	This call returns the following status values in al: 0- no error 1- invalid command 2- address mark not found 3- disk write protected 4- couldn't find sector 5- reset error 6- removed media 7- bad parameter table 8- DMA overrun 9- DMA operation crossed 64K boundary 10- illegal sector flag 11- illegal track flag 12- illegal media 13- invalid # of sectors 14- control data address mark encountered 15- DMA error 16- CRC data error 17- ECC corrected data error 32- disk controller failed 64- seek error 128- timeout error 170- drive not ready 187- undefined error 204- write error 224- status error 255- sense failure
2	al- # of sectors to read es:bx- buffer address cl- bits 0..5: sector # cl- bits 6/7- track bits 8 & 9 ch- track bits 0..7. dl- drive # (as above) dh- bits 0..5: head # dh- bits 6&7: track bits 10 & 11.	ah- return status al- burst error length carry- 0:success, 1:error	Reads the specified number of 512 byte sectors from the disk. Data read must be 64 Kbytes or less.
3	same as (2) above	same as (2) above	Writes the specified number of 512 byte sectors to the disk. Data written must not exceed 64 Kbytes in length.
4	Same as (2) above except there is no need for a buffer.	same as (2) above	Verifies the data in the specified number of 512 byte sectors on the disk.
0Ch	Same as (4) above except there is no need for a sector #	Same as (4) above	Sends the disk head to the specified track on the disk.

Table 50: Some Common Disk Subsystem BIOS Calls

AH	Input Parameters	Output Parameters	Description
0Dh	dl- drive # (80h or 81h)	ah- return status carry-0:no error 1:error	Reset the hard disk controller

Note: see appropriate BIOS documentation for additional information about disk subsystem BIOS support.

13.2.6 INT 14h - Serial I/O

Instruction: int 14h
BIOS Operation: Access the serial communications port
Parameters: ax, dx

The IBM BIOS supports up to four different serial communications ports (the hardware supports up to eight). In general, most PCs have one or two serial ports (COM1: and COM2:) installed. Int 14h supports four subfunctions- initialize, transmit a character, receive a character, and status. For all four services, the serial port number (a value in the range 0..3) is in the dx register (0=COM1:, 1=COM2:, etc.). Int 14h expects and returns other data in the al or ax register.

13.2.6.1 AH=0: Serial Port Initialization

Subfunction zero initializes a serial port. This call lets you set the baud rate, select parity modes, select the number of stop bits, and the number of bits transmitted over the serial line. These parameters are all specified by the value in the al register using the following bit encodings:

Bits Function
5..7 Select baud rate
- 000- 110 baud
- 001- 150
- 010- 300
- 011- 600
- 100- 1200
- 101- 2400
- 110- 4800
- 111- 9600

3..4 Select parity
- 00- No parity
- 01- Odd parity
- 10- No parity
- 11- Even parity

2 Stop bits
- 0-One stop bit
- 1-Two stop bits

0..1 Character Size
- 10- 7 bits
- 11- 8 bits

Although the standard PC serial port hardware supports 19,200 baud, some BIOSes may not support this speed.

Example: Initialize COM1: to 2400 baud, no parity, eight bit data, and two stop bits-

```
mov     ah, 0               ;Initialize opcode
mov     al, 10100111b       ;Parameter data.
mov     dx, 0               ;COM1: port.
int     14h
```

After the call to the initialization code, the serial port status is returned in ax (see Serial Port Status, ah=3, below).

13.2.6.2 AH=1: Transmit a Character to the Serial Port

This function transmits the character in the al register through the serial port specified in the dx register. On return, if ah contains zero, then the character was transmitted properly. If bit 7 of ah contains one, upon return, then some sort of error occurred. The remaining seven bits contain all the error statuses returned by the GetStatus call except time out error (which is returned in bit seven). If an error is reported, you should use subfunction three to get the actual error values from the serial port hardware.

Example: Transmit a character through the COM1: port

```
mov     dx, 0          ;Select COM1:
mov     al, 'a'        ;Character to transmit
mov     ah, 1          ;Transmit opcode
int     14h
test    ah, 80h        ;Check for error
jnz     SerialError
```

This function will wait until the serial port finishes transmitting the last character (if any) and then it will store the character into the transmit register.

13.2.6.3 AH=2: Receive a Character from the Serial Port

Subfunction two is used to read a character from the serial port. On entry, dx contains the serial port number. On exit, al contains the character read from the serial port and bit seven of ah contains the error status. When this routine is called, it does not return to the caller until a character is received at the serial port.

Example: Reading a character from the COM1: port

```
mov     dx, 0          ;Select COM1:
mov     ah, 2          ;Receive opcode
int     14h
test    ah, 80h        ;Check for error
jnz     SerialError

<Received character is now in AL>
```

13.2.6.4 AH=3: Serial Port Status

This call returns status information about the serial port including whether or not an error has occurred, if a character has been received in the receive buffer, if the transmit buffer is empty, and other pieces of useful information. On entry into this routine, the dx register contains the serial port number. On exit, the ax register contains the following values:

AX:	Bit Meaning
15	Time out error
14	Transmitter shift register empty
13	Transmitter holding register empty
12	Break detection error
11	Framing error
10	Parity error
9	Overrun error
8	Data available
7	Receive line signal detect
6	Ring indicator
5	Data set ready (DSR)
4	Clear to send (CTS)
3	Delta receive line signal detect
2	Trailing edge ring detector
1	Delta data set ready
0	Delta clear to send

There are a couple of useful bits, not pertaining to errors, returned in this status information. If the data available bit is set (bit #8), then the serial port has received data and you should read it from the serial port. The Transmitter holding register empty bit (bit #13) tells you if the transmit operation will be delayed while waiting for the current character to be transmitted or if the next character will be immediately transmitted. By testing these two bits, you can perform other operations while waiting for the transmit register to become available or for the receive register to contain a character.

If you're interested in serial communications, you should obtain a copy of Joe Campbell's C Programmer's Guide to Serial Communications. Although written specifically for C programmers, this book contains a lot of information useful to programmers working in any programming language. See the bibliography for more details.

13.2.7 INT 15h - Miscellaneous Services

Originally, int 15h provided cassette tape read and write services[1]. Almost immediately, everyone realized that cassettes were history, so IBM began using int 15h for many other services. Today, int 15h is used for a wide variety of function including accessing expanded memory, reading the joystick / game adapter card, and many, many other operations. Except for the joystick calls, most of these services are beyond the scope of this text. Check on the bibliography if you interested in obtaining information on this BIOS call.

13.2.8 INT 16h - Keyboard Services

Instruction: int 16h
BIOS Operation: Read a key, test for a key, or get keyboard status
Parameters: al

The IBM PC BIOS provides several function calls dealing with the keyboard. As with many of the PC BIOS routines, the number of functions has increased over the years. This section describes the three calls that were available on the original IBM PC.

1. For those who do not remember that far back, before there were hard disks people used to use only floppy disks. And before there were floppy disks, people used to use cassette tapes to store programs and data. The original IBM PC was introduced in late 1981 with a cassette port. By early 1982, no one was using cassette tape for data storage.

13.2.8.1 AH=0: Read a Key From the Keyboard

If int 16h is called with ah equal to zero, the BIOS will not return control to the caller until a key is available in the system type ahead buffer. On return, al contains the ASCII code for the key read from the buffer and ah contains the keyboard scan code. Keyboard scan codes are described in the appendices.

Certain keys on the PC's keyboard do not have any corresponding ASCII codes. The function keys, Home, PgUp, End, PgDn, the arrow keys, and the Alt keys are all good examples. When such a key is pressed, int 16h returns a zero in al and the keyboard scan code in ah. Therefore, whenever an ASCII code of zero is returned, you must check the ah register to determine which key was pressed.

Note that reading a key from the keyboard using the BIOS int 16h call does not echo the key pressed to the display. You have to call putc or use int 10h to print the character once you've read it if you want it echoed to the screen.

Example: Read a sequence of keystrokes from the keyboard until Enter is pressed.

```
ReadLoop:       mov     ah, 0           ;Read Key opcode
                int     16h
                cmp     al, 0           ;Special function?
                jz      ReadLoop        ;If so, don't echo this keystroke
                putc
                cmp     al, 0dh         ;Carriage return (ENTER)?
                jne     ReadLoop
```

13.2.8.2 AH=1: See if a Key is Available at the Keyboard

This particular int 16h subfunction allows you to check to see if a key is available in the system type ahead buffer. Even if a key is not available, control is returned (right away!) to the caller. With this call you can occasionally poll the keyboard to see if a key is available and continue processing if a key hasn't been pressed (as opposed to freezing up the computer until a key is pressed).

There are no input parameters to this function. On return, the zero flag will be clear if a key is available, set if there aren't any keys in the type ahead buffer. If a key is available, then ax will contain the scan and ASCII codes for that key. However, this function will not remove that keystroke from the typeahead buffer. Subfunction #0 must be used to remove characters. The following example demonstrates how to build a random number generator using the test keyboard function:

Example: Generating a random number while waiting for a keystroke

```
; First, clear any characters out of the type ahead buffer

ClrBuffer:      mov     ah, 1           ;Is a key available?
                int     16h
                jz      BufferIsClr     ;If not, Discontinue flushing
                mov     ah, 0           ;Flush this character from the
                int     16h             ; buffer and try again.
                jmp     ClrBuffer

BufferIsClr:    mov     cx, 0           ;Initialize "random" number.
GenRandom:      inc     cx
                mov     ah, 1           ;See if a key is available yet.
                int     16h
                jz      GenRandom
                xor     cl, ch          ;Randomize even more.
                mov     ah, 0           ;Read character from buffer
                int     16h

; Random number is now in CL, key pressed by user is in AX
```

While waiting for a key, this routine is constantly incrementing the cx register. Since human beings cannot respond rapidly (at least in terms of microseconds) the cl register will overflow many times, even for the fastest typist. As a result, cl will contain a random value since the user will not be able to control (to better than about 2ms) when a key is pressed.

13.2.8.3 AH=2: Return Keyboard Shift Key Status

This function returns the state of various keys on the PC keyboard in the al register. The values returned are as follows:

Bit	Meaning
7	Insert state (toggle by pressing INS key)
6	Caps lock (1=capslock on)
5	Num lock (1=numlock on)
4	Scroll lock (1=scroll lock on)
3	Alt (1=Alt key currently down)
2	Ctrl (1=Ctrl key currently down)
1	Left shift (1=left shift key down)
0	Right shift (1=right shift key down)

Due to a bug in the BIOS code, these bits only reflect the current status of these keys, they do not necessarily reflect the status of these keys when the next key to be read from the system type ahead buffer was depressed. In order to ensure that these status bits correspond to the state of these keys when a scan code is read from the type ahead buffer, you've got to flush the buffer, wait until a key is pressed, and then immediately check the keyboard status.

13.2.9 INT 17h - Printer Services

Instruction: int 17h
BIOS Operation: Print data and test the printer status
Parameters: ax, dx

Int 17h controls the parallel printer interfaces on the IBM PC in much the same way the int 14h controls the serial ports. Since programming a parallel port is considerably easier than controlling a serial port, using the int 17h routine is somewhat easier than using the int 14h routines.

Int 17h provides three subfunctions, specified by the value in the ah register. These subfunctions are:

0-Print the character in the AL register.
1-Initialize the printer.
2-Return the printer status.

Each of these functions is described in the following sections.

Like the serial port services, the printer port services allow you to specify which of the three printers installed in the system you wish to use (LPT1:, LPT2:, or LPT3:). The value in the dx register (0..2) specifies which printer port is to be used.

One final note- under DOS it's possible to redirect all printer output to a serial port. This is quite useful if you're using a serial printer. The BIOS printer services only talk to parallel printer adapters. If you need to send data to a serial printer using BIOS, you'll have to use int 14h to transmit the data through a serial port.

13.2.9.1 AH=0: Print a Character

If ah is zero when you call int 17h, then the BIOS will print the character in the al register. Exactly how the character code in the al register is treated is entirely up to the printer device you're using. Most printers, however, respect the printable ASCII character set and a few control characters as well. Many printers will also print all the symbols in the IBM/ASCII character set (including European, line drawing, and other special symbols). Most printers treat control characters (especially ESC sequences) in completely different manners. Therefore, if you intend to print something other than standard ASCII characters, be forewarned that your software may not work on printers other than the brand you're developing your software on.

Upon return from the int 17h subfunction zero routine, the ah register contains the current status. The values actually returned are described in the section on subfunction number two.

13.2.9.2 AH=1: Initialize Printer

Executing this call sends an electrical impulse to the printer telling it to initialize itself. On return, the ah register contains the printer status as per function number two.

13.2.9.3 AH=2: Return Printer Status

This function call checks the printer status and returns it in the ah register. The values returned are:

AH:	Bit Meaning
7	1=Printer busy, 0=printer not busy
6	1=Acknowledge from printer
5	1=Out of paper signal
4	1=Printer selected
3	1=I/O error
2	Not used
1	Not used
0	Time out error

Acknowledge from printer is, essentially, a redundant signal (since printer busy/not busy gives you the same information). As long as the printer is busy, it will not accept additional data. Therefore, calling the print character function (ah=0) will result in a delay.

The out of paper signal is asserted whenever the printer detects that it is out of paper. This signal is not implemented on many printer adapters. On such adapters it is always programmed to a logic zero (even if the printer is out of paper). Therefore, seeing a zero in this bit position doesn't always guarantee that there is paper in the machine. Seeing a one here, however, definitely means that your printer is out of paper.

The printer selected bit contains a one as long as the printer is on-line. If the user takes the printer off-line, then this bit will be cleared.

The I/O error bit contains a one if some general I/O error has occurred.

The time out error bit contains a one if the BIOS routine waited for an extended period of time for the printer to become "not busy" yet the printer remained busy.

Note that certain peripheral devices (other than printers) also interface to the parallel port, often in addition to a parallel printer. Some of these devices use the error/status signal lines to return data to the PC. The software controlling such devices often takes over the int 17h routine (via a technique we'll talk about later on) and always returns a "no error" status or "time out error" status if an error occurs on the printing device. Therefore, you should take care not to depend too heavily on these signals changing when you make the int 17h BIOS calls.

13.2.10 INT 18h - Run BASIC

Instruction:	int 18h
BIOS Operation:	Activate ROM BASIC
Parameters:	None

Executing int 18h activates the ROM BASIC interpreter in an IBM PC. However, you shouldn't use this mechanism to run BASIC since many PC compatibles do not have BASIC in ROM and the result of executing int 18h is undefined.

13.2.11 INT 19h - Reboot Computer

Instruction:	int 19h
BIOS Operation:	Restart the system
Parameters:	None

Executing this interrupt has the same effect as pressing control-alt-del on the keyboard. For obvious reasons, this interrupt service should be handled carefully!

13.2.12 INT 1Ah - Real Time Clock

Instruction:	int 1ah
BIOS Operation:	Real time clock services
Parameters:	ax, cx, dx

There are two services provided by this BIOS routine- read the clock and set the clock. The PC's real time clock maintains a counter that counts the number of 1/18ths of a second that have transpired since midnight. When you read the clock, you get the number of "ticks" which have occurred since then. When you set the clock, you specify the number of "ticks" which have occurred since midnight. As usual, the particular service is selected via the value in the ah register.

13.2.12.1 AH=0: Read the Real Time Clock

If ah = 0, then int 1ah returns a 33-bit value in al:cx:dx as follows:

Reg	Value Returned
dx	L.O. word of clock count
cx	H.O. word of clock count
al	Zero if timer has not run for more than 24 hours Non-zero otherwise.

The 32-bit value in cx:dx represents the number of 55 millisecond periods which have elapsed since midnight.

13.2.12.2 AH=1: Setting the Real Time Clock

This call allows you to set the current system time value. cx:dx contains the current count (in 55ms increments) since last midnight. Cx contains the H.O. word, dx contains the L.O. word.

13.3 An Introduction to MS-DOS™

MS-DOS provides all of the basic file manager and device manager functions required by most application programs running on an IBM PC. MS-DOS handles file I/O, character I/0, memory management, and other miscellaneous functions in a (relatively) consistent manner. If you're serious about writing software for the PC, you'll have to get real friendly with MS-DOS.

The title of this section is "An Introduction to MS-DOS". And that's exactly what it means. There is no way MS-DOS can be completely covered in a single chapter. Given all of the different books that already exist on the subject, it probably cannot even be covered by a single book (it certainly hasn't been yet. Microsoft wrote a 1,600 page book on the subject and it didn't even cover the subject fully). All this is leading up to a cop-out. There is no way this subject can be treated in more than a superficial manner in a single chapter. If you're serious about writing programs in assembly language for the PC, you'll need to complement this text with several others. Additional books on MS-DOS include: MS-DOS Programmer's Reference (also called the MS-DOS Technical Reference Manual), Peter Norton's Programmer's Guide to the IBM PC, The MS-DOS Encyclopedia, and the MS-DOS Developer's Guide. This, of course, is only a partial list of the books that are available. See the bibliography in the appendices for more details. Without a doubt, the MS-DOS Technical Reference Manual is the most important text to get your hands on. This is the official description of MS-DOS calls and parameters.

MS-DOS has a long and colorful history[2]. Throughout its lifetime, it has undergone several revisions, each purporting to be better than the last. MS-DOS' origins go all the way back to the CP/M-80 operating system written for the Intel 8080 microprocessor chip. In fact, MS-DOS v1.0 was nothing much more than a clone of CP/M-80 for Intel's 8088 microprocessor. Unfortunately, CP/M-80's file handling capabilities were horrible, to say the least. Therefore, DOS[3] improved on CP/M. New file handling capabilities, compatible with Xenix and Unix, were added to DOS, producing MS-DOS v2.0. Additional calls were added to later versions of MS-DOS. Even with the introduction of OS/2 and Windows NT (which, as this is being written, have yet to take the world by storm), Microsoft is still working on enhancements to MS-DOS which may produce even later versions.

Each new feature added to DOS introduced new DOS functions while preserving all of the functionality of the previous versions of DOS. When Microsoft rewrote the DOS file handling routines in version two, they didn't replace the old calls, they simply added new ones. While this preserved software compatibility of programs that ran under the old version of DOS, what it produced was a DOS with two sets of functionally identical, but otherwise incompatible, file services.

We're only going to concentrate on a small subset of the available DOS commands in this chapter. We're going to totally ignore those obsolete commands that have been augmented by newer, better, commands in later versions of DOS. Furthermore, we're going to skip over a description of those calls that have very little use in day to day programming. For a complete, detailed, look at the commands not covered in this chapter, you should consider the acquisition of one of the aforementioned books.

2. The MS-DOS Encyclopedia gives Microsoft's account of the history of MS-DOS. Of course, this is a one-sided presentation, but it's interesting nonetheless.
3. This text uses "DOS" to mean MS-DOS.

13.3.1 MS-DOS Calling Sequence

MS-DOS is called via the int 21h instruction. To select an appropriate DOS function, you load the ah register with a function number before issuing the int 21h instruction. Most DOS calls require other parameters as well. Generally, these other parameters are passed in the CPU's register set. The specific parameters will be discussed along with each call. Unless MS-DOS returns some specific value in a register, all of the CPU's registers are preserved across a call to DOS[4].

13.3.2 MS-DOS Character Oriented Functions

DOS provides 12 character oriented I/O calls. Most of these deal with writing and reading data to/from the keyboard, video display, serial port, and printer port. All of these functions have corresponding BIOS services. In fact, DOS usually calls the appropriate BIOS function to handle the I/O operation. However, due to DOS' redirected I/O and device driver facilities, these functions don't always call the BIOS routines. Therefore, you shouldn't call the BIOS routines (rather than DOS) simply because DOS ends up calling BIOS. Doing so may prevent your program from working with certain DOS-supported devices.

Except for function code seven, all of the following character oriented calls check the console input device (keyboard) for a control-C. If the user presses a control-C, DOS executes an int 23h instruction. Usually, this instruction will cause the program to abort and control will be returned to DOS. Keep this in mind when issuing these calls.

Microsoft considers these calls obsolete and does not guarantee they will be present in future versions of DOS. So take these first 12 routines with a rather large grain of salt. Note that the UCR Standard Library provides the functionality of many of these calls anyway, and they make the proper DOS calls, not the obsolete ones.

Table 51: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
1		al- char read	Console Input w/Echo: Reads a single character from the keyboard and displays typed character on screen.
2	dl- output char		Console Output: Writes a single character to the display.
3		al- char read	Auxiliary Input: Reads a single character from the serial port.
4	dl- output char		Auxiliary Output: Writes a single character to the output port
5	dl- output char		Printer Output: Writes a single character to the printer

4. So Microsoft claims. This may or may not be true across all versions of DOS.

Table 51: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
6	dl- output char (if not 0FFh)	al- char read (if input dl = 0FFh)	Direct Console I/O: On input, if dl contains 0FFh, this function attempts to read a character from the keyboard. If a character is available, it returns the zero flag clear and the character in al. If no character is available, it returns the zero flag set. On input, if dl contains a value other than 0FFh, this routine sends the character to the display. This routine does not do ctrl-C checking.
7		al- char read	Direct Console Input: Reads a character from the keyboard. Does not echo the character to the display. This call does not check for ctrl-C
8		al- char read	Read Keyboard w/o Echo: Just like function 7 above, except this call checks for ctrl-C.
9	ds:dx- pointer to string terminated with “$”.		Display String: This function displays the characters from location ds:dx up to (but not including) a terminating “$” character.
0Ah	ds:dx- pointer to input buffer.		Buffered Keyboard Input: This function reads a line of text from the keyboard and stores it into the input buffer pointed at by ds:dx. The first byte of the buffer must contain a count between one and 255 that contains the maximum number of allowable characters in the input buffer. This routine stores the actual number of characters read in the second byte. The actual input characters begin at the third byte of the buffer.
0Bh		al- status (0=not ready, 0FFh=ready)	Check Keyboard Status: Determines whether a character is available from the keyboard.
0Ch	al- DOS opcode 0, 1, 6, 7, or 8	al- input character if opcode 1, 6, 7, or 8.	Flush Buffer: This call empties the system type ahead buffer and then executes the DOS command specified in the al register (if al=0, no further action).

Functions 1, 2, 3, 4, 5, 9, and 0Ah are obsolete and you should not use them. Use the DOS file I/O calls instead (opcodes 3Fh and 40h).

13.3.3 MS-DOS Drive Commands

MS-DOS provides several commands that let you set the default drive, determine which drive is the default, and perform some other operations. The following table lists those functions.

Table 52: DOS Disk Drive Functions

Function # (AH)	Input Parameters	Output Parameters	Description
0Dh			Reset Drive: Flushes all file buffers to disk. Generally called by ctrl-C handlers or sections of code that need to guaranteed file consistency because an error may occur.
0Eh	dl- drive number	al- number of logical drives	Set Default Drive: sets the DOS default drive to the specified value (0=A, 1=B, 2=C, etc.). Returns the number of logical drives in the system, although they may not be contiguous from 0-al.
19H		al- default drive number	Get Default Drive: Returns the current system default drive number (0=A, 1=B, 2=C, etc.).
1Ah	ds:dx- Disk Transfer Area address.		Set Disk Transfer Area Address: Sets the address that MS-DOS uses for obsolete file I/O and Find First/Find Next commands.
1Bh		al- sectors/cluster cx- bytes/sector dx- # of clusters ds:bx - points at media descriptor byte	Get Default Drive Data: Returns information about the disk in the default drive. Also see function 36h. Typical values for the media descriptor byte include: 0F0h- 3.5” 0F8h- Hard disk 0F9h- 720K 3.5” or 1.2M 5.25” 0FAh- 320K 5.25” 0FBh- 640K 3.5” 0FCh- 180K 5.25” 0FDh- 360K 5.25: 0FEh- 160K 5.25” 0FFh- 320K 5.25”
1Ch	dl- drive number	See above	Get Drive Data: same as above except you can specify the drive number in the dl register (0=default, 1=A, 2=B, 3=C, etc.).

Table 52: DOS Disk Drive Functions

Function # (AH)	Input Parameters	Output Parameters	Description
1Fh		al- contains 0FFh if error, 0 if no error. ds:bx- ptr to DPB	Get Default Disk Parameter Block (DPB): If successful, this function returns a pointer to the following structure: Drive (byte) - Drive number (0-A, 1=B, etc.). Unit (byte) - Unit number for driver. SectorSize (word) - # bytes / sector. ClusterMask (byte) - sectors / cluster minus one. Cluster2 (byte) - $2^{clusters / sector}$ FirstFAT (word) - Address of sector where FAT starts. FATCount (byte) - # of FATs. RootEntries (word) - # of entries in root directory. FirstSector (word) - first sector of first cluster. MaxCluster (word) - # of clusters on drive, plus one. FATsize (word) - # of sectors for FAT. DirSector (word) - first sector containing directory. DriverAdrs (dword) - address of device driver. Media (byte) - media descriptor byte. FirstAccess (byte) - set if there has been an access to drive. NextDPB (dword) - link to next DPB in list. NextFree (word) - last allocated cluster. FreeCnt (word) - number of free clusters.
2Eh	al- verify flag (0=no verify, 1=verify on).		Set / Reset Verify Flag: Turns on and off write verification. Usually off since this is a slow operation, but you can turn it on when performing critical I / O.
2Fh		es:bx- pointer to DTA	Get Disk Transfer Area Address: Returns a pointer to the current DTA in es:bx..
32h	dl- drive number.	Same as 1Fh	Get DPB: Same as function 1Fh except you get to specify the driver number (0=default, 1=A, 2=B, 3=C, etc.).
33h	al- 05 (subfunction code)	dl- startup drive #.	Get Startup Drive: Returns the number of the drive used to boot DOS (1=A, 2=B, 3=C, etc.).
36h	dl- drive number.	ax- sectors / cluster bx- available clusters cx- bytes / sector dx- total clusters	Get Disk Free Space: Reports the amount of free space. This call supersedes calls 1Bh and 1Ch that only support drives up to 32Mbytes. This call handles larger drives. You can compute the amount of free space (in bytes) by bx*ax*cx. If an error occurs, this call returns 0FFFFh in ax.
54h		al- verify state.	Get Verify State: Returns the current state of the write verify flag (al=0 if off, al=1 if on).

13.3.4 MS-DOS "Obsolete" Filing Calls

DOS functions 0Fh - 18h, 1Eh, 20h-24h, and 26h - 29h are the functions left over from the days of CP / M-80. In general, you shouldn't bother at all with these calls since

MS-DOS v2.0 and later provides a much better way to accomplish the operations performed by these calls.

13.3.5 MS-DOS Date and Time Functions

The MS-DOS date and time functions return the current date and time based on internal values maintained by the real time clock (RTC). Functions provided by DOS include reading and setting the date and time. These date and time values are used to perform date and time stamping of files when files are created on the disk. Therefore, if you change the date or time, keep in mind that it will have an effect on the files you create thereafter. Note that the UCR Standard Library also provides a set of date and time functions which, in many cases, are somewhat easier to use than these DOS calls.

Table 53: Date and Time Functions

Function # (AH)	Input Parameters	Output Parameters	Description
2Ah		al- day (0=Sun, 1=Mon, etc.). cx- year dh- month (1=Jan, 2=Feb, etc.). dl- Day of month (1-31).	Get Date: returns the current MS-DOS date.
2Bh	cx- year (1980 - 2099) dh- month (1-12) dl- day (1-31)		Set Date: sets the current MS-DOS date.
2CH		ch- hour (24hr fmt) cl- minutes dh- seconds dl- hundredths	Get Time: reads the current MS-DOS time. Note that the hundredths of a second field has a resolution of $^{1}/_{18}$ second.
2Dh	ch- hour cl- minutes dh- seconds dl- hundredths		Set Time: sets the current MS-DOS time.

13.3.6 MS-DOS Memory Management Functions

MS-DOS provides three memory management functions- allocate, deallocate, and resize (modify). For most programs, these three memory allocation calls are not used. When DOS executes a program, it gives all of the available memory, from the start of that program to the end of RAM, to the executing process. Any attempt to allocate memory without first giving unused memory back to the system will produce an “insufficient memory” error.

Sophisticated programs which terminate and remain resident, run other programs, or perform complex memory management tasks, may require the use of these memory management functions. Generally these types of programs immediately deallocate all of the memory that they don’t use and then begin allocating and deallocating storage as they see fit.

Since these are complex functions, they shouldn't be used unless you have a very specific purpose for them. Misusing these commands may result in loss of system memory that can be reclaimed only by rebooting the system. Each of the following calls returns the error status in the carry flag. If the carry is clear on return, then the operation was completed successfully. If the carry flag is set when DOS returns, then the ax register contains one of the following error codes:

7- Memory control blocks destroyed
8- Insufficient memory
9- Invalid memory block address

Additional notes about these errors will be discussed as appropriate.

13.3.6.1 Allocate Memory

Function (ah): 48h
Entry parameters: bx- Requested block size (in paragraphs)
Exit parameters: If no error (carry clear):
ax:0 points at allocated memory block

If an error (carry set):
bx- maximum possible allocation size
ax- error code (7 or 8)

This call is used to allocate a block of memory. On entry into DOS, bx contains the size of the requested block in paragraphs (groups of 16 bytes). On exit, assuming no error, the ax register contains the segment address of the start of the allocated block. If an error occurs, the block is not allocated and the ax register is returned containing the error code. If the allocation request failed due to insufficient memory, the bx register is returned containing the maximum number of paragraphs actually available.

13.3.6.2 Deallocate Memory

Function (ah): 49h
Entry parameters: es:0- Segment address of block to be deallocated
Exit parameters: If the carry is set, ax contains the error code (7,9)

This call is used to deallocate memory allocated via function 48h above. The es register cannot contain an arbitrary memory address. It must contain a value returned by the allocate memory function. You cannot use this call to deallocate a portion of an allocated block. The modify allocation function is used for that operation.

13.3.6.3 Modify Memory Allocation

Function (ah): 4Ah
Entry parameters: es:0- address of block to modify allocation size
bx- size of new block
Exit parameters: If the carry is set, then
ax contains the error code 7, 8, or 9
bx contains the maximum size possible (if error 8)

This call is used to change the size of an allocated block. On entry, es must contain the segment address of the allocated block returned by the memory allocation function. Bx must contain the new size of this block in paragraphs. While you can almost always reduce the size of a block, you cannot normally increase the size of a block if other blocks have been allocated after the block being modified. Keep this in mind when using this function.

13.3.6.4 Advanced Memory Management Functions

The MS-DOS 58h opcode lets programmers adjust MS-DOS' memory allocation strategy and control the use of upper memory blocks (UMBs). There are four subfunctions to this call, with the subfunction value appearing in the al register. The following table describes these calls:

Table 54: Advanced Memory Management Functions

Function # (AH)	Input Parameters	Output Parameters	Description
58h	al-0	ax- strategy	Get Allocation Strategy: Returns the current allocation strategy in ax (see table below for details).
58h	al-1 bx- strategy		Set Allocation Strategy: Sets the MS-DOS allocation strategy to the value specified in bx (see the table below for details).
58H	al- 2	al- link flag	Get Upper Memory Link: Returns true/false (1/0) in al to determine whether a program can allocate memory in the upper memory blocks.
58h	al- 3 bx- link flag (0=no link, 1=link okay).		Set Upper Memory Link: Links or unlinks the upper memory area. When linked, an application can allocate memory from the UMB (using the normal DOS allocate call).

Table 55: Memory Allocation Strategies

Value	Name	Description
0	First Fit Low	Search conventional memory for the first free block of memory large enough to satisfy the allocation request. This is the default case.
1	Best Fit Low	Search conventional memory for the smallest block large enough to satisfy the request.
2	Last Fit Low	Search conventional memory from the highest address downward for the first block large enough to satisfy the request.
80h	First Fit High	Search high memory, then conventional memory, for the first available block that can satisfy the allocation request.
81h	Best Fit High	Search high memory, then conventional memory for the smallest block large enough to satisfy the allocation request.
82h	Last Fit High	Search high memory from high addresses to low, then conventional memory from high addresses to low, for the first block large enough to satisfy the request.
40h	First Fit Highonly	Search high memory only for the first block large enough to satisfy the request.
41h	Best Fit Highonly	Search high memory only for the smallest block large enough to satisfy the request.

Table 55: Memory Allocation Strategies

Value	Name	Description
42h	Last Fit Highonly	Search high memory only, from the end of memory downward, for the first block large enough to satisfy the request.

These different allocation strategies can have an impact on system performance. For an analysis of different memory management strategies, please consult a good operating systems theory text.

13.3.7 MS-DOS Process Control Functions

DOS provides several services dealing with loading, executing, and terminating programs. Many of these functions have been rendered obsolete by later versions of DOS. There are three[5] functions of general interest- program termination, terminate and stay resident, and execute a program. These three functions will be discussed in the following sections.

13.3.7.1 Terminate Program Execution

Function (ah): 4Ch
Entry parameters: al- return code
Exit parameters: Does not return to your program

This is the function call normally used to terminate your program. It returns control to the calling process (normally, but not necessarily, DOS). A return code can be passed to the calling process in the al register. Exactly what meaning this return code has is entirely up to you. This return code can be tested with the DOS "IF ERRORLEVEL return code" command in a DOS batch file. All files opened by the current process will be automatically closed upon program termination.

Note that the UCR Standard Library function "ExitPgm" is simply a macro which makes this particular DOS call. This is the normal way of returning control back to MS-DOS or some other program which ran the currently active application.

13.3.7.2 Terminate, but Stay Resident

Function (ah): 31h
Entry parameters: al- return code
dx- memory size, in paragraphs
Exit parameters: does not return to your program

This function also terminates program execution, but upon returning to DOS, the memory in use by the process is not returned to the DOS free memory pool. Essentially, the program remains in memory. Programs which remain resident in memory after returning to DOS are often called TSRs (terminate and stay resident programs).

When this command is executed, the dx register contains the number of memory paragraphs to leave around in memory. This value is measured from the beginning of the "program segment prefix", a segment marking the start of your file in memory. The address of the PSP (program segment prefix) is passed to your program in the ds register

5. Actually, there are others. See the DOS technical reference manual for more details. We will only consider these three here.

when your program is first executed. You'll have to save this value if your program is a TSR[6].

Programs that terminate and stay resident need to provide some mechanism for restarting. Once they return to DOS they cannot normally be restarted. Most TSRs patch into one of the interrupt vectors (such as a keyboard, printer, or serial interrupt vector) in order to restart whenever some hardware related event occurs (such as when a key is pressed). This is how "pop-up" programs like SmartKey work.

Generally, TSR programs are pop-ups or special device drivers. The TSR mechanism provides a convenient way for you to load your own routines to replace or augment BIOS' routines. Your program loads into memory, patches the appropriate interrupt vector so that it points at an interrupt handler internal to your code, and then terminates and stays resident. Now, when the appropriate interrupt instruction is executed, your code will be called rather than BIOS'.

There are far too many details concerning TSRs including compatibility issues, DOS re-entrancy issues, and how interrupts are processed, to be considered here. Additional details will appear in a later chapter.

13.3.7.3 Execute a Program

Function (ah):	40h
Entry parameters:	ds:dx- pointer to pathname of program to execute
	es:bx- Pointer to parameter block
	al- 0=load and execute, 1=load only, 3=load overlay.
Exit parameters:	If carry is set, ax contains one of the following error codes:
	1- invalid function
	2- file not found
	5- access denied
	8- not enough memory
	10- bad environment
	11- bad format

The execute (exec) function is an extremely complex, but at the same time, very useful operation. This command allows you to load or load and execute a program off of the disk drive. On entry into the exec function, the ds:dx registers contain a pointer to a zero terminated string containing the name of the file to be loaded or executed, es:bx points at a parameter block, and al contains zero or one depending upon whether you want to load and execute a program or simply load it into memory. On return, if the carry is clear, then DOS properly executed the command. If the carry flag is set, then DOS encountered an error while executing the command.

The filename parameter can be a full pathname including drive and subdirectory information. "B:\DIR1\DIR2\MYPGM.EXE" is a perfectly valid filename (remember, however, it must be zero terminated). The segmented address of this pathname is passed in the ds:dx registers.

The es:bx registers point at a parameter block for the exec call. This parameter block takes on three different forms depending upon whether a program is being loaded and executed (al=0), just loaded into memory (al=1), or loaded as an overlay (al=3).

If al=0, the exec call loads and executes a program. In this case the es:bx registers point at a parameter block containing the following values:

Offset	Description
0	A word value containing the segment address of the default environment (usually this is set to zero which implies the use of the standard DOS environment).
2	Double word pointer containing the segment address of a command line string.

6. DOS also provides a call which will return the PSP for your program.

6	Double word pointer to default FCB at address 5Ch
0Ah	Double word pointer to default FCB at address 6Ch

The environment area is a set of strings containing default pathnames and other information (this information is provided by DOS using the PATH, SET, and other DOS commands). If this parameter entry contains zero, then exec will pass the standard DOS environment on to the new procedure. If non-zero, then this parameter contains the segment address of the environment block that your process is passing on to the program about to be executed. Generally, you should store a zero at this address.

The pointer to the command string should contain the segmented address of a length prefixed string which is also terminated by a carriage return character (the carriage return character is not figured into the length of the string). This string corresponds to the data that is normally typed after the program name on the DOS command line. For example, if you're executing the linker automatically, you might pass a command string of the following form:

```
CmdStr      byte       16,"MyPgm+Routines /m",0dh
```

The second item in the parameter block must contain the segmented address of this string.

The third and fourth items in the parameter block point at the default FCBs. FCBs are used by the obsolete DOS filing commands, so they are rarely used in modern application programs. Since the data structures these two pointers point at are rarely used, you can point them at a group of 20 zeros.

Example: Format a floppy disk in drive A: using the FORMAT.EXE command

```
                  mov      ah, 4Bh
                  mov      al, 0
                  mov      dx, seg PathName
                  mov      ds, dx
                  lea      dx, PathName
                  mov      bx, seg ParmBlock
                  mov      es, bx
                  lea      bx, ParmBlock
                  int      21h
                    .
                    .
                    .
PathName          byte     'C:\DOS\FORMAT.EXE',0
ParmBlock         word     0                        ;Default environment
                  dword    CmdLine                  ;Command line string
                  dword    Dummy,Dummy              ;Dummy FCBs

CmdLine           byte     3,' A:',0dh
Dummy             byte     20 dup (?)
```

MS-DOS versions earlier than 3.0 do not preserve any registers except cs:ip when you execute the exec call. In particular, ss:sp is not preserved. If you're using DOS v2.x or earlier, you'll need to use the following code:

```
;Example: Format a floppy disk in drive A: using the FORMAT.EXE command

                  <push any registers you need preserved>

                  mov      cs:SS_Save, ss           ;Save SS:SP to a location
                  mov      cs:SP_Save, sp           ; we have access to later.
                  mov      ah, 4Bh                  ;EXEC DOS opcode.
                  mov      al, 0                    ;Load and execute.
                  mov      dx, seg PathName         ;Get filename into DS:DX.
                  mov      ds, dx
                  lea      dx, PathName
                  mov      bx, seg ParmBlock        ;Point ES:BX at parameter
                  mov      es, bx                   ; block.
                  lea      bx, ParmBlock
                  int      21h
                  mov      ss, cs:SS_Save           ;Restore SS:SP from saved
                  mov      sp, cs:SP_Save           ; locations.
```

```
        <Restore registers pushed onto the stack>
                      .
                      .
                      .
SS_Save         word    ?
SP_Save         word    ?
                      .
                      .
                      .
PathName        byte    'C:\DOS\FORMAT.EXE',0
ParmBlock       word    0                       ;Default environment
                dword   CmdLine                 ;Command line string
                dword   Dummy,Dummy;Dummy       ;FCBs
CmdLine         byte    3,' A:',0dh
Dummy           byte    20 dup (?)
```

SS_Save and SP_Save must be declared inside your code segment. The other variables can be declared anywhere.

The exec command automatically allocates memory for the program being executed. If you haven't freed up unused memory before executing this command, you may get an insufficient memory error. Therefore, you should use the DOS deallocate memory command to free up unused memory before attempting to use the exec command.

If al=1 when the exec function executes, DOS will load the specified file but will not execute it. This function is generally used to load a program to execute into memory but give the caller control and let the caller start that code. When this function call is made, es:bx points at the following parameter block:

Offset	Description
0	Word value containing the segment address of the environment block for the new process. If you want to use the parent process' environment block set this word to zero.
2	Dword pointer to the command tail for this operation. The command tail is the command line string which will appear at location PSP:80 (See "The Program Segment Prefix (PSP)" on page 739 and "Accessing Command Line Parameters" on page 742).
6	Address of default FCB #1. For most programs, this should point at a block of 20 zeros (unless, of course, you're running a program which uses FCBs.).
0Ah	Address of default FCB #2. Should also point at a block of 20 zeros.
0Eh	SS:SP value. You must load these four bytes into SS and SP before starting the application.
12h	CS:IP value. These four bytes contain the starting address of the program.

The SSSP and CSIP fields are output values. DOS fills in the fields and returns them in the load structure. The other fields are all inputs which you must fill in before calling the exec function with al=1.

When you execute the exec command with al=-3, DOS simply loads an *overlay* into memory. Overlays generally consist of a single code segment which contains some functions you want to execute. Since you are not creating a new process, the parameter block for this type of load is much simpler than for the other two types of load operations. On entry, es:bx must point at the following parameter block in memory:

Offset	Description
0	Word value containing the segment address of where this file is going to be loaded into memory. The file will be loaded at offset zero within this segment.
2	Word value containing a relocation factor for this file.

Unlike the load and execute functions, the overlay function does not automatically allocate storage for the file being loaded. Your program has to allocate sufficient storage and then pass the address of this storage block to the exec command (though the parameter block above). Only the segment address of this block is passed to the exec command, the offset is always assumed to be zero. The relocation factor should also contain the segment address for ".EXE" files. For ".COM" files, the relocation factor parameter should be zero.

The overlay command is quite useful for loading overlays from disk into memory. An overlay is a segment of code which resides on the disk drive until the program actually needs to execute its code. Then the code is loaded into memory and executed. Overlays can reduce the amount of memory your program takes up by allowing you to reuse the same portion of memory for different overlay procedures (clearly, only one such procedure can be active at any one time). By placing seldom-used code and initialization code into overlay files, you can help reduce the amount of memory used by your program file. One word of caution, however, managing overlays is a very complex task. This is not something a beginning assembly language programmer would want to tackle right away. When loading a file into memory (as opposed to loading and executing a file), DOS does not scramble all of the registers, so you needn't take the extra care necessary to preserve the ss:sp and other registers.

The MS-DOS Encyclopedia contains an excellent description of the use of the exec function.

13.3.8 MS-DOS "New" Filing Calls

Starting with DOS v2.0, Microsoft introduced a set of file handling procedures which (finally) made disk file access bearable under MS-DOS. Not only bearable, but actually easy to use! The following sections describe the use of these commands to access files on a disk drive.

File commands which deal with filenames (Create, Open, Delete, Rename, and others) are passed the address of a zero-terminated pathname. Those that actually open a file (Create and Open) return a file handle as the result (assuming, of course, that there wasn't an error). This file handle is used with other calls (read, write, seek, close, etc.) to gain access to the file you've opened. In this respect, a file handle is not unlike a file variable in Pascal. Consider the following Microsoft / Turbo Pascal code:

```
program DemoFiles; var F:TEXT;
begin
        assign(f,'FileName.TXT');
        rewrite(f);
        writeln(f,'Hello there');
        close(f);
end.
```

The file variable "f" is used in this Pascal example in much the same way that a file handle is used in an assembly language program – to gain access to the file that was created in the program.

All the following DOS filing commands return an error status in the carry flag. If the carry flag is clear when DOS returns to your program, then the operation was completed successfully. If the carry flag is set upon return, then some sort of error has occurred and the AX register contains the error number. The actual error return values will be discussed along with each function in the following sections.

13.3.8.1 Open File

Function (ah): 3Dh
Entry parameters:
al- file access value
 0- File opened for reading
 1- File opened for writing
 2- File opened for reading and writing
ds:dx- Point at a zero terminated string containing the filename.
Exit parameters: If the carry is set, ax contains one of the following error codes:
 2- File not found

4- Too many open files
5- Access denied
12- Invalid access
If the carry is clear, ax contains the file handle value assigned by DOS.

A file must be opened before you can access it. The open command opens a file that already exists. This makes it quite similar to Pascal's Reset procedure. Attempting to open a file that doesn't exist produces an error. Example:

```
lea     dx, Filename            ;Assume DS points at segment
mov     ah, 3dh                 ; of filename
mov     al, 0                   ;Open for reading.
int     21h
jc      OpenError
mov     FileHandle, ax
```

If an error occurs while opening a file, the file will not be opened. You should always check for an error after executing a DOS open command, since continuing to operate on the file which hasn't been properly opened will produce disastrous consequences. Exactly how you handle an open error is up to you, but at the very least you should print an error message and give the user the opportunity to specify a different filename.

If the open command completes without generating an error, DOS returns a file handle for that file in the ax register. Typically, you should save this value away somewhere so you can use it when accessing the file later on.

13.3.8.2 Create File

Function (ah):	3Ch
Entry parameters:	ds:dx- Address of zero terminated pathname
	cx- File attribute
Exit parameters:	If the carry is set, ax contains one of the following error codes:
	3- Path not found
	4- Too many open files
	5- Access denied
	If the carry is clear, ax is returned containing the file handle

Create opens a new file for output. As with the OPEN command, ds:dx points at a zero terminated string containing the filename. Since this call creates a new file, DOS assumes that you're opening the file for writing only. Another parameter, passed in cx, is the initial file attribute settings. The L.O. six bits of cx contain the following values:

Bit	Meaning if equal to one
0	File is a Read-Only file
1	File is a hidden file
2	File is a system file
3	File is a volume label name
4	File is a subdirectory
5	File has been archived

In general, you shouldn't set any of these bits. Most normal files should be created with a file attribute of zero. Therefore, the cx register should be loaded with zero before calling the create function.

Upon exit, the carry flag is set if an error occurs. The "Path not found" error requires some additional explanation. This error is generated, not if the file isn't found (which would be most of the time since this command is typically used to create a new file), but if a subdirectory in the pathname cannot be found.

If the carry flag is clear when DOS returns to your program, then the file has been properly opened for output and the ax register contains the file handle for this file.

13.3.8.3 Close File

Function (ah): 3Eh
Entry parameters: bx- File Handle
Exit parameters: If the carry flag is set, ax contains 6, the only possible error, which is an invalid handle error.

This call is used to close a file opened with the Open or Create commands above. It is passed the file handle in the bx register and, assuming the file handle is valid, closes the specified file.

You should close all files your program uses as soon as you're through with them to avoid disk file corruption in the event the user powers the system down or resets the machine while your files are left open.

Note that quitting to DOS (or aborting to DOS by pressing control-C or control-break) automatically closes all open files. However, you should never rely on this feature since doing so is an extremely poor programming practice.

13.3.8.4 Read From a File

Function (ah): 3Fh
Entry parameters: bx- File handle
cx- Number of bytes to read
ds:dx- Array large enough to hold bytes read
Exit parameters: If the carry flag is set, ax contains one of the following error codes
5- Access denied
6- Invalid handle
If the carry flag is clear, ax contains the number of bytes actually read from the file.

The read function is used to read some number of bytes from a file. The actual number of bytes is specified by the cx register upon entry into DOS. The file handle, which specifies the file from which the bytes are to be read, is passed in the bx register. The ds:dx register contains the address of a buffer into which the bytes read from the file are to be stored.

On return, if there wasn't an error, the ax register contains the number of bytes actually read. Unless the end of file (EOF) was reached, this number will match the value passed to DOS in the cx register. If the end of file has been reached, the value return in ax will be somewhere between zero and the value passed to DOS in the cx register. *This is the only test for the EOF condition.*

Example: This example opens a file and reads it to the EOF

```
                mov     ah, 3dh         ;Open the file
                mov     al, 0           ;Open for reading
                lea     dx, Filename ;Presume DS points at filename
                int     21h             ; segment.
                jc      BadOpen
                mov     FHndl, ax       ;Save file handle

LP:             mov     ah,3fh          ;Read data from the file
                lea     dx, Buffer      ;Address of data buffer
                mov     cx, 1           ;Read one byte
                mov     bx, FHndl       ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx          ;EOF reached?
                jne     EOF
                mov     al, Buffer      ;Get character read
                putc                    ;Print it
                jmp     LP              ;Read next byte

EOF:            mov     bx, FHndl
                mov     ah, 3eh         ;Close file
```

```
int     21h
jc      CloseError
```

This code segment will read the entire file whose (zero-terminated) filename is found at address "Filename" in the current data segment and write each character in the file to the standard output device using the UCR StdLib putc routine. Be forewarned that one-character-at-a-time I/O such as this is extremely slow. We'll discuss better ways to quickly read a file a little later in this chapter.

13.3.8.5 Write to a File

Function (ah): 40h
Entry parameters: bx- File handle
cx- Number of bytes to write
ds:dx- Address of buffer containing data to write
Exit parameters: If the carry is set, ax contains one of the following error codes
5- Accessed denied
6- Invalid handle
If the carry is clear on return, ax contains the number of bytes actually written to the file.

This call is almost the converse of the read command presented earlier. It writes the specified number of bytes at ds:dx to the file rather than reading them. On return, if the number of bytes written to the file is not equal to the number originally specified in the cx register, the disk is full and this should be treated as an error.

If cx contains zero when this function is called, DOS will truncate the file to the current file position (i.e., all data following the current position in the file will be deleted).

13.3.8.6 Seek (Move File Pointer)

Function (ah): 42h Entry parameters:
al- Method of moving
0- Offset specified is from the beginning of the file.
1- Offset specified is distance from the current file pointer.
2- The pointer is moved to the end of the file minus the specified offset.
bx- File handle.
cx:dx- Distance to move, in bytes.
Exit parameters: If the carry is set, ax contains one of the following error codes
1- Invalid function
6- Invalid handle
If the carry is clear, dx:ax contains the new file position

This command is used to move the file pointer around in a random access file. There are three methods of moving the file pointer, an absolute distance within the file (if al=0), some positive distance from the current file position (if al=1), or some distance from the end of the file (if al=2). If AL doesn't contain 0, 1, or 2, DOS will return an invalid function error. If this call is successfully completed, the next byte read or written will occur at the specified location.

Note that DOS treats cx:dx as an unsigned integer. Therefore, a single seek command cannot be used to move backwards in the file. Instead, method #0 must be used to position the file pointer at some absolute position in the file. If you don't know where you currently are and you want to move back 256 bytes, you can use the following code:

```
mov     ah, 42h                 ;Seek command
mov     al, 1                   ;Move from current location
xor     cx, cx                  ;Zero out CX and DX so we
xor     dx, dx                  ; stay right here
```

```
mov     bx, FileHandle
int     21h
jc      SeekError
sub     ax, 256                 ;DX:AX now contains the
sbb     dx, 0                   ; current file position, so
mov     cx, dx                  ; compute a location 256
mov     dx, ax                  ; bytes back.
mov     ah, 42h
mov     al, 0                   ;Absolute file position
int     21h                     ;BX still contains handle.
```

13.3.8.7 Set Disk Transfer Address (DTA)

Function (ah):	1Ah Entry parameters:
	ds:dx- Pointer to DTA
Exit parameters:	None

This command is called "Set Disk Transfer Address" because it was (is) used with the original DOS v1.0 file functions. We wouldn't normally consider this function except for the fact that it is also used by functions 4Eh and 4Fh (described next) to set up a pointer to a 43-byte buffer area. If this function isn't executed before executing functions 4Eh or 4Fh, DOS will use the default buffer space at PSP:80h.

13.3.8.8 Find First File

Function (ah):	4Eh
Entry parameters:	cx- Attributes
	ds:dx- Pointer to filename
Exit parameters:	If carry is set, ax contains one of the following error codes
	2- File not found
	18- No more files

The Find First File and Find Next File (described next) functions are used to search for files specified using ambiguous file references. An ambiguous file reference is any filename containing the "*" and "?" wildcard characters. The Find First File function is used to locate the first such filename within a specified directory, the Find Next File function is used to find successive entries in the directory.

Generally, when an ambiguous file reference is provided, the Find First File command is issued to locate the first occurrence of the file, and then a loop is used, calling Find Next File, to locate all other occurrences of the file within that loop until there are no more files (error #18). Whenever Find First File is called, it sets up the following information at the DTA:

Offset	Description
0	Reserved for use by Find Next File
21	Attribute of file found
22	Time stamp of file
24	Date stamp of file
26	File size in bytes
30	Filename and extension (zero terminated)

(The offsets are decimal)

Assuming Find First File doesn't return some sort of error, the name of the first file matching the ambiguous file description will appear at offset 30 in the DTA.

Note: if the specified pathname doesn't contain any wildcard characters, then Find First File will return the exact filename specified, if it exists. Any subsequent call to Find Next File will return an error.

The cx register contains the search attributes for the file. Normally, cx should contain zero. If non-zero, Find First File (and Find Next File) will include file names which have the specified attributes as well as all normal file names.

13.3.8.9 Find Next File

Function (ah): 4Fh
Entry parameters: none
Exit parameters: If the carry is set, then there aren't any more files and ax will be returned containing 18.

The Find Next File function is used to search for additional file names matching an ambiguous file reference after a call to Find First File. The DTA must point at a data record set up by the Find First File function.

Example: The following code lists the names of all the files in the current directory that end with ".EXE". Presumably, the variable "DTA" is in the current data segment:

```
                mov     ah, 1Ah                 ;Set DTA
                lea     dx, DTA
                int     21h
                xor     cx, cx                  ;No attributes.
                lea     dx, FileName
                mov     ah, 4Eh                 ;Find First File
                int     21h
                jc      NoMoreFiles             ;If error, we're done
DirLoop:        lea     si, DTA+30              ;Address of filename
                cld
PrtName:        lodsb
                test    al, al                  ;Zero byte?
                jz      NextEntry
                putc                            ;Print this character
                jmp     PrtName

NextEntry:      mov     ah, 4Fh                 ;Find Next File
                int     21h
                jnc     DirLoop                 ;Print this name
```

13.3.8.10 Delete File

Function (ah): 41h
Entry parameters: ds:dx- Address of pathname to delete
Exit parameters: If carry set, ax contains one of the following error codes
2- File not found
5- Access denied

This function will delete the specified file from the directory. The filename must be an unambiguous filename (i.e., it cannot contain any wildcard characters).

13.3.8.11 Rename File

Function (ah): 56h Entry parameters:
ds:dx- Pointer to pathname of existing file
es:di- Pointer to new pathname
Exit parameters: If carry set, ax contains one of the following error codes
2- File not found
5- Access denied
17- Not the same device

This command serves two purposes: it allows you to rename one file to another and it allows you to move a file from one directory to another (as long as the two subdirectories are on the same disk).

Example: Rename “MYPGM.EXE” to “YOURPGM.EXE”

```
; Assume ES and DS both point at the current data segment
; containing the filenames.

                  lea      dx, OldName
                  lea      di, NewName
                  mov      ah, 56h
                  int      21h
                  jc       BadRename
                            .
                            .
                            .
OldName           byte     "MYPGM.EXE",0
NewName           byte     "YOURPGM.EXE",0

Example #2: Move a filename from one directory to another:
; Assume ES and DS both point at the current data segment
; containing the filenames.

                  lea      dx, OldName
                  lea      di, NewName
                  mov      ah, 56h
                  int      21h
                  jc       BadRename
                            .
                            .
                            .
OldName           byte     "\DIR1\MYPGM.EXE",0
NewName           byte     "\DIR2\MYPGM.EXE",0
```

13.3.8.12 Change/Get File Attributes

Function (ah): 43h
Entry parameters: al- Subfunction code
0- Return file attributes in cx
1- Set file attributes to those in cx
cx- Attribute to be set if AL=01
ds:dx- address of pathname
Exit parameters: If carry set, ax contains one of the following error codes:
1- Invalid function
3- Pathname not found
5- Access denied
If the carry is clear and the subfunction was zero cx will contain the file’s attributes.

This call is useful for setting/resetting and reading a file’s attribute bits. It can be used to set a file to read-only, set/clear the archive bit, or otherwise mess around with the file attributes.

13.3.8.13 Get/Set File Date and Time

Function (ah): 57h
Entry parameters: al- Subfunction code
0- Get date and time
1- Set date and time
bx- File handle
cx- Time to be set (if AL=01)
dx- Date to be set (if AL=01)

Exit parameters: If carry set, ax contains one of the following error codes
1- Invalid subfunction
6- Invalid handle
If the carry is clear, cx/dx is set to the time/date if al=00

This call sets the "last-write" date/time for the specified file. The file must be open (using open or create) before using this function. The date will not be recorded until the file is closed.

13.3.8.14 Other DOS Calls

The following tables briefly list many of the other DOS calls. For more information on the use of these DOS functions consult the Microsoft MS-DOS Programmer's Reference or the MS-DOS Technical Reference.

Table 56: Miscellaneous DOS File Functions

Function # (AH)	Input Parameters	Output Parameters	Description
39h	ds:dx- pointer to zero terminated pathname.		Create Directory: Creates a new directory with the specified name.
3Ah	ds:dx- pointer to zero terminated pathname.		Remove Directory: Deletes the directory with the specified pathname. Error if directory is not empty or the specified directory is the current directory.
3Bh	ds:dx- pointer to zero terminated pathname.		Change Directory: Changes the default directory to the specified pathname.
45h	bx- file handle	ax- new handle	Duplicate File Handle: creates a copy of a file handle so a program can access a file using two separate file variables. This allows the program to close the file with one handle and continue accessing it with the other.
46h	bx- file handle cx- duplicate handle		Force Duplicate File Handle: Like function 45h above, except you specify which handle (in cx) you want to refer to the existing file (specified by bx).
47h	ds:si- pointer to buffer dl- drive		Get Current Directory: Stores a string containing the current pathname (terminated with a zero) starting at location ds:si. These registers must point at a buffer containing at least 64 bytes. The dl register specifies the drive number (0=default, 1=A, 2=B, 3=C, etc.).
5Ah	cx- attributes ds:dx- pointer to temporary path.	ax- handle	Create Temporary File: Creates a file with a unique name in the directory specified by the zero terminated string at which ds:dx points. There must be at least 13 zero bytes beyond the end of the pathname because this function will store the generated filename at the end of the pathname. The attributes are the same as for the Create call.

Table 56: Miscellaneous DOS File Functions

Function # (AH)	Input Parameters	Output Parameters	Description
5Bh	cx- attributes ds:dx- pointer to zero terminated pathname.	ax- handle	Create New File: Like the create call, but this call insists that the file not exist. It returns an error if the file exists (rather than deleting the old file).
67h	bx- handles		Set Maximum Handle Count: This function sets the maximum number of handles a program can use at any one given time.
68h	bx- handle		Commit File: Flushes all data to a file without closing it, ensuring that the file's data is current and consistent.

Table 57: Miscellaneous DOS Functions

Function # (AH)	Input Parameters	Output Parameters	Description
25h	al- interrupt # ds:dx- pointer to interrupt service routine.		Set Interrupt Vector: Stores the specified address in ds:dx into the interrupt vector table at the entry specified by the al register.
30h		al- major version ah- minor version bh- Version flag bl:cx- 24 bit serial number	Get Version Number: Returns the current version number of DOS (or value set by SETVER).
33h	al- 0	dl- break flag (0=off, 1=on)	Get Break Flag: Returns the status of the DOS break flag. If on, MS-DOS checks for ctrl-C when processing any DOS command; if off, MS-DOS only checks on functions 1-0Ch.
33h	al- 1 dl- break flag.		Set Break Flag: Sets the MS-DOS break flag according to the value in dl (see function above for details).
33h	al- 6	bl- major version bh- minor version dl- revision dh- version flags	Get MS-DOS Version: Returns the "real" version number, not the one set by the SETVER command. Bits three and four of the version flags are one if DOS is in ROM or DOS is in high memory, respectively.
34h		es:bx- pointer to InDOS flag.	Get InDOS Flag Address: Returns the address of the InDOS flag. This flag helps prevent reentrancy in TSR applications
35h	al- interrupt #	es:bx- pointer to interrupt service routine.	Get Interrupt Vector: Returns a pointer to the interrupt service routine for the specified interrupt number. See function 25h above for more details.
44h	al- subcode Other parameters!		Device Control: This is a whole family of additional DOS commands to control various devices. See the DOS programmer's reference manual for more details.

Table 57: Miscellaneous DOS Functions

Function # (AH)	Input Parameters	Output Parameters	Description
4Dh		al- return value ah- termination method	Get Child Program Return Value: Returns the last result code from a child program in al. The ah register contains the termination method, which is one of the following values: 0-normal, 1-ctrl-C, 2-critical device error, 3-terminate and stay resident.
50h	bx- PSP address		Set PSP Address: Set DOS' current PSP address to the value specified in the bx register.
51h		bx- PSP address	Get PSP Address: Returns a pointer to the current PSP in the bx register.
59h		ax- extended error bh- error class bl- error action ch- error location	Get Extended Error: Returns additional information when an error occurs on a DOS call. See the DOS programmer's guide for more details on these errors and how to handle them.
5Dh	al- 0Ah ds:si- pointer to extended error structure.		Set Extended Error: copies the data from the extended error structure to DOS' internal record.

In addition to the above commands, there are several additional DOS calls that deal with networks and international character sets. See the MS-DOS reference for more details.

13.3.9 File I/O Examples

Of course, one of the main reasons for making calls to DOS is to manipulate files on a mass storage device. The following examples demonstrate some uses of character I/O using DOS.

13.3.9.1 Example #1: A Hex Dump Utility

This program dumps a file in hexadecimal format. The filename must be hard coded into the file (see "Accessing Command Line Parameters" later in this chapter).

```
                  include    stdlib.a
                  includelib stdlib.lib

cseg              segment  byte public 'CODE'
                  assume   cs:cseg, ds:dseg, es:dseg, ss:sseg

MainPgm           proc     far

; Properly set up the segment registers:

                  mov      ax, seg dseg
                  mov      ds, ax
                  mov      es, ax
                  mov      ah, 3dh
                  mov      al, 0                  ;Open file for reading
                  lea      dx, Filename           ;File to open
                  int      21h
                  jnc      GoodOpen
```

```
                print
                byte    'Cannot open file, aborting program...',cr,0
                jmp     PgmExit

GoodOpen:       mov     FileHandle, ax          ;Save file handle
                mov     Position, 0             ;Initialize file pos counter
ReadFileLp:     mov     al, byte ptr Position
                and     al, 0Fh                 ;Compute (Position MOD 16)
                jnz     NotNewLn                ;Start new line each 16 bytes
                putcr
                mov     ax, Position            ;Print offset into file
                xchg    al, ah
                puth
                xchg    al, ah
                puth
                print
                byte    ': ',0

NotNewLn:       inc     Position                ;Increment character count
                mov     bx, FileHandle
                mov     cx, 1                   ;Read one byte
                lea     dx, buffer              ;Place to store that byte
                mov     ah, 3Fh                 ;Read operation
                int     21h
                jc      BadRead
                cmp     ax, 1                   ;Reached EOF?
                jnz     AtEOF
                mov     al, Buffer              ;Get the character read and
                puth                            ; print it in hex
                mov     al, ' '                 ;Print a space between values
                putc
                jmp     ReadFileLp

BadRead:        print
                byte    cr, lf
                byte    'Error reading data from file, aborting'
                byte    cr,lf,0

AtEOF:          mov     bx, FileHandle          ;Close the file
                mov     ah, 3Eh
                int     21h

PgmExit:        ExitPgm
MainPgm         endp

cseg            ends
dseg            segment   byte public 'data'

Filename        byte      'hexdump.asm',0               ;Filename to dump
FileHandle      word      ?
Buffer          byte      ?
Position        word      0

dseg            ends

sseg            segment   byte stack 'stack'
stk             word      0ffh dup (?)
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end       MainPgm
```

13.3.9.2 Example #2: Upper Case Conversion

The following program reads one file, converts all the lower case characters to upper case, and writes the data to a second output file.

```
                include    stdlib.a
                includelib stdlib.lib
```

```
cseg            segment     byte public 'CODE'
                assume      cs:cseg, ds:dseg, es:dseg, ss:sseg

MainPgm         proc        far

; Properly set up the segment registers:

                mov      ax, seg dseg
                mov      ds, ax
                mov      es, ax

;-----------------------------------------------------------------
;
; Convert UCCONVRT.ASM to uppercase
;
; Open input file:

                mov      ah, 3dh
                mov      al, 0                  ;Open file for reading
                lea      dx, Filename           ;File to open
                int      21h
                jnc      GoodOpen
                print
                byte     'Cannot open file, aborting program...',cr,lf,0
                jmp      PgmExit

GoodOpen:       mov      FileHandle1, ax        ;Save input file handle

; Open output file:

                mov      ah, 3Ch                ;Create file call
                mov      cx, 0                  ;Normal file attributes
                lea      dx, OutFileName        ;File to open
                int      21h
                jnc      GoodOpen2
                print
                byte     'Cannot open output file, aborting program...'
                byte     cr,lf,0
                mov      ah, 3eh                ;Close input file
                mov      bx, FileHandle1
                int      21h
                jmp      PgmExit                ;Ignore any error.

GoodOpen2:      mov      FileHandle2, ax        ;Save output file handle

ReadFileLp:     mov      bx, FileHandle1
                mov      cx, 1                  ;Read one byte
                lea      dx, buffer             ;Place to store that byte
                mov      ah, 3Fh                ;Read operation
                int      21h
                jc       BadRead
                cmp      ax, 1                  ;Reached EOF?
                jz       ReadOK
                jmp      AtEOF

ReadOK:         mov      al, Buffer             ;Get the character read and
                cmp      al, 'a'                ; convert it to upper case
                jb       NotLower
                cmp      al, 'z'
                ja       NotLower
                and      al, 5fh                ;Set Bit #5 to zero.
NotLower:       mov      Buffer, al

; Now write the data to the output file

                mov      bx, FileHandle2
                mov      cx, 1                  ;Read one byte
                lea      dx, buffer             ;Place to store that byte
                mov      ah, 40h                ;Write operation
                int      21h
                jc       BadWrite
                cmp      ax, 1                  ;Make sure disk isn't full
                jz       ReadFileLp

BadWrite:       print
```

```
                byte     cr, lf
                byte     'Error writing data to file, aborting operation'
                byte     cr,lf,0
                jmp      short AtEOF

BadRead:        print
                byte     cr, lf
                byte     'Error reading data from file, aborting '
                byte     'operation',cr,lf,0

AtEOF:          mov      bx, FileHandle1        ;Close the file
                mov      ah, 3Eh
                int      21h
                mov      bx, FileHandle2
                mov      ah, 3eh
                int      21h

;----------------------------------------------------------------

PgmExit:        ExitPgm
MainPgm         endp
cseg            ends

dseg            segment  byte public 'data'

Filename        byte       'ucconvrt.asm',0            ;Filename to convert
OutFileName     byte       'output.txt',0              ;Output filename
FileHandle1     word       ?
FileHandle2     word       ?
Buffer          byte       ?
Position        word       0

dseg            ends

sseg            segment    byte stack 'stack'
stk             word       0ffh dup (?)
sseg            ends

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end      MainPgm
```

13.3.10 Blocked File I/O

The examples in the previous section suffer from a major drawback, they are extremely slow. The performance problems with the code above are entirely due to DOS. Making a DOS call is not, shall we say, the fastest operation in the world. Calling DOS every time we want to read or write a single character from/to a file will bring the system to its knees. As it turns out, it doesn't take (practically) any more time to have DOS read or write two characters than it does to read or write one character. Since the amount of time we (usually) spend processing the data is negligible compared to the amount of time DOS takes to return or write the data, reading two characters at a time will essentially double the speed of the program. If reading two characters doubles the processing speed, how about reading four characters? Sure enough, it almost quadruples the processing speed. Likewise processing ten characters at a time almost increases the processing speed by an order of magnitude. Alas, this progression doesn't continue forever. There comes a point of diminishing returns- when it takes far too much memory to justify a (very) small improvement in performance (keeping in mind that reading 64K in a single operation requires a 64K memory buffer to hold the data). A good compromise is 256 or 512 bytes. Reading more data doesn't really improve the performance much, yet a 256 or 512 byte buffer is easier to deal with than larger buffers.

Reading data in groups or blocks is called *blocked I/O*. Blocked I/O is often one to two orders of magnitude faster than single character I/O, so obviously you should use blocked I/O whenever possible.

There is one minor drawback to blocked I/O-- it's a little more complex to program than single character I/O. Consider the example presented in the section on the DOS read command:

Example: This example opens a file and reads it to the EOF

```
                mov     ah, 3dh             ;Open the file
                mov     al, 0               ;Open for reading
                lea     dx, Filename        ;Presume DS points at
filename
                int     21h                 ; segment
                jc      BadOpen
                mov     FHndl, ax           ;Save file handle

LP:             mov     ah,3fh              ;Read data from the file
                lea     dx, Buffer          ;Address of data buffer
                mov     cx, 1               ;Read one byte
                mov     bx, FHndl           ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx              ;EOF reached?
                jne     EOF
                mov     al, Buffer          ;Get character read
                putc                        ;Print it (IOSHELL call)
                jmp     LP                  ;Read next byte

EOF:            mov     bx, FHndl
                mov     ah, 3eh             ;Close file
                int     21h
                jc      CloseError
```

There isn't much to this program at all. Now consider the same example rewritten to use blocked I/O:

Example: This example opens a file and reads it to the EOF using blocked I/O

```
                mov     ah, 3dh             ;Open the file
                mov     al, 0               ;Open for reading
                lea     dx, Filename        ;Presume DS points at
filename
                int     21h                 ; segment
                jc      BadOpen
                mov     FHndl, ax           ;Save file handle

LP:             mov     ah,3fh              ;Read data from the file
                lea     dx, Buffer          ;Address of data buffer
                mov     cx, 256             ;Read 256 bytes
                mov     bx, FHndl           ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx;EOF reached?
                jne     EOF
                mov     si, 0               ;Note: CX=256 at this point.
PrtLp:          mov     al, Buffer[si]      ;Get character read
                putc                        ;Print it
                inc     si
                loop    PrtLp
                jmp     LP                  ;Read next block

; Note, just because the number of bytes read doesn't equal 256,
; don't get the idea we're through, there could be up to 255 bytes
; in the buffer still waiting to be processed.

EOF:            mov     cx, ax
                jcxz    EOF2          ;If CX is zero, we're really done.
                mov     si, 0         ;Process the last block of data read
Finis:          mov     al, Buffer[si]; from the file which contains
                putc                  ; 1..255 bytes of valid data.
                inc     si
                loop    Finis

EOF2:           mov     bx, FHndl
                mov     ah, 3eh ;Close file
```

```
                int     21h
                jc      CloseError
```

This example demonstrates one major hassle with blocked I/O – when you reach the end of file, you haven't necessarily processed all of the data in the file. If the block size is 256 and there are 255 bytes left in the file, DOS will return an EOF condition (the number of bytes read don't match the request). In this case, we've still got to process the characters that were read. The code above does this in a rather straight-forward manner, using a second loop to finish up when the EOF is reached. You've probably noticed that the two print loops are virtually identical. This program can be reduced in size somewhat using the following code which is only a little more complex:

Example: This example opens a file and reads it to the EOF using blocked I/O

```
                mov     ah, 3dh                 ;Open the file
                mov     al, 0                   ;Open for reading
                lea     dx, Filename            ;Presume DS points at
filename
                int     21h                     ; segment.
                jc      BadOpen
                mov     FHndl, ax               ;Save file handle

LP:             mov     ah,3fh                  ;Read data from the file
                lea     dx, Buffer              ;Address of data buffer
                mov     cx, 256                 ;Read 256 bytes
                mov     bx, FHndl               ;Get file handle value
                int     21h
                jc      ReadError
                mov     bx, ax                  ;Save for later
                mov     cx, ax
                jcxz    EOF
                mov     si, 0                   ;Note: CX=256 at this point.
PrtLp:          mov     al, Buffer[si]          ;Get character read
                putc                            ;Print it
                inc     si
                loop    PrtLp
                cmp     bx, 256                 ;Reach EOF yet?
                je      LP

EOF:            mov     bx, FHndl
                mov     ah, 3eh                 ;Close file
                int     21h
                jc      CloseError
```

Blocked I/O works best on sequential files. That is, those files opened only for reading or writing (no seeking). When dealing with random access files, you should read or write whole records at one time using the DOS read/write commands to process the whole record. This is still considerably faster than manipulating the data one byte at a time.

13.3.11 The Program Segment Prefix (PSP)

When a program is loaded into memory for execution, DOS first builds up a program segment prefix immediately before the program is loaded into memory. This PSP contains lots of information, some of it useful, some of it obsolete. Understanding the layout of the PSP is essential for programmers designing assembly language programs.

The PSP is 256 bytes long and contains the following information:

Offset	Length	Description
0	2	An INT 20h instruction is stored here
2	2	Program ending address
4	1	Unused, reserved by DOS
5	5	Call to DOS function dispatcher
0Ah	4	Address of program termination code

0Eh	4	Address of break handler routine
12h	4	Address of critical error handler routine
16h	22	Reserved for use by DOS
2Ch	2	Segment address of environment area
2Eh	34	Reserved by DOS
50h	3	INT 21h, RETF instructions
53h	9	Reserved by DOS
5Ch	16	Default FCB #1
6Ch	20	Default FCB #2
80h	1	Length of command line string
81h	127	Command line string

Note: locations 80h..FFh are used for the default DTA.

Most of the information in the PSP is of little use to a modern MS-DOS assembly language program. Buried in the PSP, however, are a couple of gems that are worth knowing about. Just for completeness, however, we'll take a look at all of the fields in the PSP.

The first field in the PSP contains an int 20h instruction. Int 20h is an obsolete mechanism used to terminate program execution. Back in the early days of DOS v1.0, your program would execute a jmp to this location in order to terminate. Nowadays, of course, we have DOS function 4Ch which is much easier (and safer) than jumping to location zero in the PSP. Therefore, this field is obsolete.

Field number two contains a value which points at the last paragraph allocated to your program By subtracting the address of the PSP from this value, you can determine the amount of memory allocated to your program (and quit if there is insufficient memory available).

The third field is the first of many "holes" left in the PSP by Microsoft. Why they're here is anyone's guess.

The fourth field is a call to the DOS function dispatcher. The purpose of this (now obsolete) DOS calling mechanism was to allow some additional compatibility with CP/M-80 programs. For modern DOS programs, there is absolutely no need to worry about this field.

The next three fields are used to store special addresses during the execution of a program. These fields contain the default terminate vector, break vector, and critical error handler vectors. These are the values normally stored in the interrupt vectors for int 22h, int 23h, and int 24h. By storing a copy of the values in the vectors for these interrupts, you can change these vectors so that they point into your own code. When your program terminates, DOS restores those three vectors from these three fields in the PSP. For more details on these interrupt vectors, please consult the DOS technical reference manual.

The eighth field in the PSP record is another reserved field, currently unavailable for use by your programs.

The ninth field is another real gem. It's the address of the environment strings area. This is a two-byte pointer which contains the segment address of the environment storage area. The environment strings always begin with an offset zero within this segment. The environment string area consists of a sequence of zero-terminated strings. It uses the following format:

$string_1$ 0 $string_2$ 0 $string_3$ 0 ... 0 $string_n$ 0 0

That is, the environment area consists of a list of zero terminated strings, the list itself being terminated by a string of length zero (i.e., a zero all by itself, or two zeros in a row, however you want to look at it). Strings are (usually) placed in the environment area via DOS commands like PATH, SET, etc. Generally, a string in the environment area takes the form

```
name = parameters
```

For example, the “SET IPATH=C:\ASSEMBLY\INCLUDE” command copies the string “IPATH=C:\ASSEMBLY\INCLUDE” into the environment string storage area.

Many languages scan the environment storage area to find default filename paths and other pieces of default information set up by DOS. Your programs can take advantage of this as well.

The next field in the PSP is another block of reserved storage, currently undefined by DOS.

The 11th field in the PSP is another call to the DOS function dispatcher. Why this call exists (when the one at location 5 in the PSP already exists and nobody really uses either mechanism to call DOS) is an interesting question. In general, this field should be ignored by your programs.

The 12th field is another block of unused bytes in the PSP which should be ignored.

The 13th and 14th fields in the PSP are the default FCBs (File Control Blocks). File control blocks are another archaic data structure carried over from CP/M-80. FCBs are used only with the obsolete DOS v1.0 file handling routines, so they are of little interest to us. We’ll ignore these FCBs in the PSP.

Locations 80h through the end of the PSP contain a very important piece of information- the command line parameters typed on the DOS command line along with your program’s name. If the following is typed on the DOS command line:

```
MYPGM parameter1, parameter2
```

the following is stored into the command line parameter field:

```
23, “ parameter1, parameter2”, 0Dh
```

Location 80h contains 23_{10}, the length of the parameters following the program name. Locations 81h through 97h contain the characters making up the parameter string. Location 98h contains a carriage return. Notice that the carriage return character is not figured into the length of the command line string.

Processing the command line string is such an important facet of assembly language programming that this process will be discussed in detail in the next section.

Locations 80h..FFh in the PSP also comprise the default DTA. Therefore, if you don’t use DOS function 1Ah to change the DTA and you execute a FIND FIRST FILE, the filename information will be stored starting at location 80h in the PSP.

One important detail we’ve omitted until now is exactly how you access data in the PSP. Although the PSP is loaded into memory immediately before your program, that doesn’t necessarily mean that it appears 100h bytes before your code. Your data segments may have been loaded into memory before your code segments, thereby invalidating this method of locating the PSP. The segment address of the PSP is passed to your program in the ds register. To store the PSP address away in your data segment, your programs should begin with the following code:

```
push    ds                      ;Save PSP value
mov     ax, seg DSEG            ;Point DS and ES at our data
mov     ds, ax                  ; segment.
mov     es, ax
pop     PSP                     ;Store PSP value into “PSP”
                                ; variable.
 .
 .
 .
```

Another way to obtain the PSP address, in DOS 5.0 and later, is to make a DOS call. If you load ah with 51h and execute an int 21h instruction, MS-DOS will return the segment address of the current PSP in the bx register.

There are lots of tricky things you can do with the data in the PSP. Peter Norton’s Programmer’s Guide to the IBM PC lists all kinds of tricks. Such operations won’t be discussed here because they’re a little beyond the scope of this manual.

13.3.12 Accessing Command Line Parameters

Most programs like MASM and LINK allow you to specify command line parameters when the program is executed. For example, by typing

```
ML MYPGM.ASM
```

you can instruct MASM to assemble MYPGM without any further intervention from the keyboard. "MYPGM.ASM;" is a good example of a command line parameter.

When DOS' COMMAND.COM command interpreter parses your command line, it copies most of the text following the program name to location 80h in the PSP as described in the previous section. For example, the command line above will store the following at PSP:80h

```
11, " MYPGM.ASM", 0Dh
```

The text stored in the command line tail storage area in the PSP is usually an exact copy of the data appearing on the command line. There are, however, a couple of exceptions. First of all, I/O redirection parameters are not stored in the input buffer. Neither are command tails following the pipe operator ("|"). The other thing appearing on the command line which is absent from the data at PSP:80h is the program name. This is rather unfortunate, since having the program name available would allow you to determine the directory containing the program. Nevertheless, there is lots of useful information present on the command line.

The information on the command line can be used for almost any purpose you see fit. However, most programs expect two types of parameters in the command line parameter buffer-- filenames and switches. The purpose of a filename is rather obvious, it allows a program to access a file without having to prompt the user for the filename. Switches, on the other hand, are arbitrary parameters to the program. By convention, switches are preceded by a slash or hyphen on the command line.

Figuring out what to do with the information on the command line is called *parsing* the command line. Clearly, if your programs are to manipulate data on the command line, you've got to parse the command line within your code.

Before a command line can be parsed, each item on the command line has to be separated out apart from the others. That is, each word (or more properly, lexeme[7]) has to be identified in the command line. Separation of lexemes on a command line is relatively easy, all you've got to do is look for sequences of delimiters on the command line. Delimiters are special symbols used to separate tokens on the command line. DOS supports six different delimiter characters: space, comma, semicolon, equal sign, tab, or carriage return.

Generally, any number of delimiter characters may appear between two tokens on a command line. Therefore, all such occurrences must be skipped when scanning the command line. The following assembly language code scans the entire command line and prints all of the tokens that appear thereon:

```
                include     stdlib.a
                includelib  stdlib.lib

cseg            segment     byte public 'CODE'
                assume      cs:cseg, ds:dseg, es:dseg, ss:sseg

; Equates into command line-

CmdLnLen        equ     byte ptr es:[80h]       ;Command line length
CmdLn           equ     byte ptr es:[81h]       ;Command line data

tab             equ     09h

MainPgm         proc    far

; Properly set up the segment registers:
```

7. Many programmers use the term "token" rather than lexeme. Technically, a token is a different entity.

```
                push    ds                      ;Save PSP
                mov     ax, seg dseg
                mov     ds, ax
                pop     PSP

;-----------------------------------------------------------------

                print
                byte    cr,lf
                byte    'Items on this line:',cr,lf,lf,0

                mov     es, PSP                 ;Point ES at PSP
                lea     bx, CmdLn               ;Point at command line
PrintLoop:      print
                byte    cr,lf,'Item: ',0
                call    SkipDelimiters          ;Skip over leading delimiters
PrtLoop2:       mov     al, es:[bx]             ;Get next character
                call    TestDelimiter           ;Is it a delimiter?
                jz      EndOfToken              ;Quit this loop if it is
                putc                            ;Print char if not.
                inc     bx                      ;Move on to next character
                jmp     PrtLoop2

EndOfToken:     cmp     al, cr                  ;Carriage return?
                jne     PrintLoop               ;Repeat if not end of line

                print
                byte    cr,lf,lf
                byte    'End of command line',cr,lf,lf,0
                ExitPgm
MainPgm         endp

; The following subroutine sets the zero flag if the character in
; the AL register is one of DOS' six delimiter characters,
; otherwise the zero flag is returned clear. This allows us to use
; the JE/JNE instructions afterwards to test for a delimiter.

TestDelimiter   proc    near
                cmp     al, ' '
                jz      ItsOne
                cmp     al,','
                jz      ItsOne
                cmp     al,Tab
                jz      ItsOne
                cmp     al,';'
                jz      ItsOne
                cmp     al,'='
                jz      ItsOne
                cmp     al, cr
ItsOne:         ret
TestDelimiter   endp

; SkipDelimiters skips over leading delimiters on the command
; line. It does not, however, skip the carriage return at the end
; of a line since this character is used as the terminator in the
; main program.

SkipDelimiters  proc    near
                dec     bx                      ;To offset INC BX below
SDLoop:          inc    bx                      ;Move on to next character.
                mov     al, es:[bx]             ;Get next character
                cmp     al, 0dh                 ;Don't skip if CR.
                jz      QuitSD
                call    TestDelimiter           ;See if it's some other
                jz      SDLoop                  ; delimiter and repeat.
QuitSD:         ret
SkipDelimiters  endp

cseg            ends

dseg            segment byte public 'data'

PSP             word    ?                       ;Program segment prefix
dseg            ends
```

```
sseg            segment  byte stack 'stack'
stk             word     0ffh dup (?)
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end      MainPgm
```

Once you can scan the command line (that is, separate out the lexemes), the next step is to parse it. For most programs, parsing the command line is an extremely trivial process. If the program accepts only a single filename, all you've got to do is grab the first lexeme on the command line, slap a zero byte onto the end of it (perhaps moving it into your data segment), and use it as a filename. The following assembly language example modifies the hex dump routine presented earlier so that it gets its filename from the command line rather than hard-coding the filename into the program:

```
                include    stdlib.a
                includelib stdlib.lib
cseg            segment    byte public 'CODE'
                assume     cs:cseg, ds:dseg, es:dseg, ss:sseg

; Note CR and LF are already defined in STDLIB.A

tab             equu     09h

MainPgm         proc     far

; Properly set up the segment registers:

                mov      ax, seg dseg
                mov      es, ax                 ;Leave DS pointing at PSP

;---------------------------------------------------------------
;
; First, parse the command line to get the filename:

                mov      si, 81h                ;Pointer to command line
                lea      di, FileName           ;Pointer to FileName buffer
SkipDelimiters:
                lodsb                           ;Get next character
                call     TestDelimiter
                je       SkipDelimiters

; Assume that what follows is an actual filename

                dec      si                     ;Point at 1st char of name
GetFName:       lodsb
                cmp      al, 0dh
                je       GotName
                call     TestDelimiter
                je       GotName
                stosb                           ;Save character in file name
                jmp      GetFName

; We're at the end of the filename, so zero-terminate it as
; required by DOS.

GotName:        mov      byte ptr es:[di], 0
                mov      ax, es                 ;Point DS at DSEG
                mov      ds, ax

; Now process the file

                mov      ah, 3dh
                mov      al, 0                  ;Open file for reading
                lea      dx, Filename           ;File to open
                int      21h
                jnc      GoodOpen
                print
                byte     'Cannot open file, aborting program...',cr,0
                jmp      PgmExit

GoodOpen:       mov      FileHandle, ax         ;Save file handle
```

```
                mov     Position, 0             ;Initialize file position
ReadFileLp:     mov     al, byte ptr Position
                and     al, 0Fh                 ;Compute (Position MOD 16)
                jnz     NotNewLn                ;Every 16 bytes start a line
                putcr
                mov     ax, Position            ;Print offset into file
                xchg    al, ah
                puth
                xchg    al, ah
                puth
                print
                byte    ': ',0

NotNewLn:       inc     Position                ;Increment character count
                mov     bx, FileHandle
                mov     cx, 1                   ;Read one byte
                lea     dx, buffer              ;Place to store that byte
                mov     ah, 3Fh                 ;Read operation
                int     21h
                jc      BadRead
                cmp     ax, 1                   ;Reached EOF?
                jnz     AtEOF
                mov     al, Buffer              ;Get the character read and
                puth                            ; print it in hex
                mov     al, ' '                 ;Print a space between values
                putc
                jmp     ReadFileLp

BadRead:        print
                byte    cr, lf
                byte    'Error reading data from file, aborting.'
                byte    cr,lf,0

AtEOF:          mov     bx, FileHandle          ;Close the file
                mov     ah, 3Eh
                int     21h

;---------------------------------------------------------------

PgmExit:        ExitPgm
MainPgm         endp

TestDelimiter   proc    near
                cmp     al, ' '
                je      xit
                cmp     al, ','
                je      xit
                cmp     al, Tab
                je      xit
                cmp     al, ';'
                je      xit
                cmp     al, '='
xit:            ret
TestDelimiter   endp
cseg            ends

dseg            segment byte public 'data'

PSP             word    ?
Filename        byte    64 dup (0)              ;Filename to dump
FileHandle      word    ?
Buffer          byte    ?
Position        word    0

dseg            ends

sseg            segment byte stack 'stack'
stk             word    0ffh dup (?)
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
```

```
                end     MainPgm
```

The following example demonstrates several concepts dealing with command line parameters. This program copies one file to another. If the "/U" switch is supplied (somewhere) on the command line, all of the lower case characters in the file are converted to upper case before being written to the destination file. Another feature of this code is that it will prompt the user for any missing filenames, much like the MASM and LINK programs will prompt you for filename if you haven't supplied any.

```
                include    stdlib.a
                includelib stdlib.lib

cseg            segment    byte public 'CODE'
                assume     cs:cseg, ds:nothing, es:dseg, ss:sseg

; Note: The constants CR (0dh) and LF (0ah) appear within the
; stdlib.a include file.

tab             equu    09h

MainPgm         proc    far

; Properly set up the segment registers:

                mov     ax, seg dseg
                mov     es, ax                  ;Leave DS pointing at PSP

;---------------------------------------------------------------

; First, parse the command line to get the filename:

                mov     es:GotName1, 0          ;Init flags that tell us if
                mov     es:GotName2, 0          ; we've parsed the filenames
                mov     es:ConvertLC,0          ; and the "/U" switch.

; Okay, begin scanning and parsing the command line

                mov     si, 81h                 ;Pointer to command line
SkipDelimiters:
                lodsb                           ;Get next character
                call    TestDelimiter
                je      SkipDelimiters

; Determine if this is a filename or the /U switch

                cmp     al, '/'
                jnz     MustBeFN

; See if it's "/U" here-

                lodsb
                and     al, 5fh                 ;Convert "u" to "U"
                cmp     al, 'U'
                jnz     NotGoodSwitch
                lodsb                           ;Make sure next char is
                cmp     al, cr                  ; a delimiter of some sort
                jz      GoodSwitch
                call    TestDelimiter
                jne     NotGoodSwitch

; Okay, it's "/U" here.

GoodSwitch:     mov     es:ConvertLC, 1         ;Convert LC to UC
                dec     si                      ;Back up in case it's CR
                jmp     SkipDelimiters          ;Move on to next item.

; If a bad switch was found on the command line, print an error
; message and abort-

NotGoodSwitch:
                print
                byte    cr,lf
                byte    'Illegal switch, only "/U" is allowed!',cr,lf
                byte    'Aborting program execution.',cr,lf,0
                jmp     PgmExit

; If it's not a switch, assume that it's a valid filename and
; handle it down here-
```

```
MustBeFN:       cmp     al, cr                  ;See if at end of cmd line
                je      EndOfCmdLn

; See if it's filename one, two, or if too many filenames have been
; specified-

                cmp     es:GotName1, 0
                jz      Is1stName
                cmp     es:GotName2, 0
                jz      Is2ndName

; More than two filenames have been entered, print an error message
; and abort.

                print
                byte    cr,lf
                byte    'Too many filenames specified.',cr,lf
                byte    'Program aborting...',cr,lf,lf,0
                jmp     PgmExit

; Jump down here if this is the first filename to be processed-

Is1stName:      lea     di, FileName1
                mov     es:GotName1, 1
                jmp     ProcessName

Is2ndName:      lea     di, FileName2
                mov     es:GotName2, 1
ProcessName:
                stosb                           ;Store away character in name
                lodsb                           ;Get next char from cmd line
                cmp     al, cr
                je      NameIsDone
                call    TestDelimiter
                jne     ProcessName

NameIsDone:     mov     al, 0                   ;Zero terminate filename
                stosb
                dec     si                      ;Point back at previous char
                jmp     SkipDelimiters          ;Try again.

; When the end of the command line is reached, come down here and
; see if both filenames were specified.

                assume    ds:dseg

EndOfCmdLn:     mov     ax, es                  ;Point DS at DSEG
                mov     ds, ax

; We're at the end of the filename, so zero-terminate it as
;  required by DOS.

GotName:        mov     ax, es                  ;Point DS at DSEG
                mov     ds, ax

; See if the names were supplied on the command line.
; If not, prompt the user and read them from the keyboard

                cmp     GotName1, 0             ;Was filename #1 supplied?
                jnz     HasName1
                mov     al, '1'                 ;Filename #1
                lea     si, Filename1
                call    GetName                 ;Get filename #1

HasName1:       cmp     GotName2, 0             ;Was filename #2 supplied?
                jnz     HasName2
                mov     al, '2'                 ;If not, read it from kbd.
                lea     si, FileName2
                call    GetName

; Okay, we've got the filenames, now open the files and copy the
; source file to the destination file.

HasName2        mov     ah, 3dh
                mov     al, 0                   ;Open file for reading
                lea     dx, Filename1           ;File to open
```

```
                int     21h
                jnc     GoodOpen1

                print
                byte    'Cannot open file, aborting program...',cr,lf,0
                jmp     PgmExit

; If the source file was opened successfully, save the file handle.

GoodOpen1:      mov     FileHandle1, ax         ;Save file handle

; Open (CREATE, actually) the second file here.

                mov     ah, 3ch                 ;Create file
                mov     cx, 0                   ;Standard attributes
                lea     dx, Filename2           ;File to open
                int     21h
                jnc     GoodCreate

; Note: the following error code relies on the fact that DOS
; automatically closes any open source files when the program
; terminates.

                print
                byte    cr,lf
                byte    'Cannot create new file, aborting operation'
                byte    cr,lf,lf,0
                jmp     PgmExit

GoodCreate:     mov     FileHandle2, ax         ;Save file handle

; Now process the files

CopyLoop:       mov     ah, 3Fh                 ;DOS read opcode
                mov     bx, FileHandle1         ;Read from file #1
                mov     cx, 512                 ;Read 512 bytes
                lea     dx, buffer              ;Buffer for storage
                int     21h
                jc      BadRead
                mov     bp, ax                  ;Save # of bytes read

                cmp     ConvertLC,0             ;Conversion option active?
                jz      NoConversion

; Convert all LC in buffer to UC-

                mov     cx, 512
                lea     si, Buffer
                mov     di, si
ConvertLC2UC:
                lodsb
                cmp     al, 'a'
                jb      NoConv
                cmp     al, 'z'
                ja      NoConv
                and     al, 5fh
NoConv:         stosb
                loop    ConvertLC2UC

NoConversion:
                mov     ah, 40h                 ;DOS write opcode
                mov     bx, FileHandle2         ;Write to file #2
                mov     cx, bp                  ;Write however many bytes
                lea     dx, buffer              ;Buffer for storage
                int     21h
                jc      BadWrite
                cmp     ax, bp                  ;Did we write all of the
                jnz     jDiskFull               ; bytes?
                cmp     bp, 512                 ;Were there 512 bytes read?
                jz      CopyLoop
                jmp     AtEOF
jDiskFull:      jmp     DiskFull

; Various error messages:

BadRead:        print
```

```
                byte    cr,lf
                byte    'Error while reading source file, aborting '
                byte    'operation.',cr,lf,0
                jmp     AtEOF

BadWrite:       print
                byte    cr,lf
                byte    'Error while writing destination file, aborting'
                byte    ' operation.',cr,lf,0
                jmp     AtEOF

DiskFull:       print
                byte    cr,lf
                byte    'Error, disk full.  Aborting operation.',cr,lf,0

AtEOF:          mov     bx, FileHandle1         ;Close the first file
                mov     ah, 3Eh
                int     21h
                mov     bx, FileHandle2         ;Close the second file
                mov     ah, 3Eh
                int     21h

PgmExit:        ExitPgm
MainPgm         endp

TestDelimiter   proc    near
                cmp     al, ' '
                je      xit
                cmp     al, ','
                je      xit
                cmp     al, Tab
                je      xit
                cmp     al, ';'
                je      xit
                cmp     al, '='
xit:            ret
TestDelimiter   endp

; GetName- Reads a filename from the keyboard.  On entry, AL
; contains the filename number and DI points at the buffer in ES
; where the zero-terminated filename must be stored.

GetName         proc    near
                print
                byte    'Enter filename #',0
                putc
                mov     al, ':'
                putc
                gets
                ret
GetName         endp
cseg            ends

dseg            segment byte public 'data'

PSP             word    ?
Filename1       byte    128 dup (?);Source filename
Filename2       byte    128 dup (?);Destination filename
FileHandle1     word    ?
FileHandle2     word    ?
GotName1        byte    ?
GotName2        byte    ?
ConvertLC       byte    ?
Buffer          byte    512 dup (?)

dseg            ends

sseg            segment   byte stack 'stack'
stk             word    0ffh dup (?)
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end     MainPgm
```

As you can see, there is more effort expended processing the command line parameters than actually copying the files!

13.3.13 ARGC and ARGV

The UCR Standard Library provides two routines, argc and argv, which provide easy access to command line parameters. Argc (*argument count*) returns the number of items on the command line. Argv (*argument vector*) returns a pointer to a specific item in the command line.

These routines break up the command line into lexemes using the standard delimiters. As per MS-DOS convention, argc and argv treat any string surrounded by quotation marks on the command line as a single command line item.

Argc will return in cx the number of command line items. Since MS-DOS does not include the program name on the command line, this count does not include the program name either. Furthermore, redirection operands (“>filename” and “<filename”) and items to the right of a pipe (“| command”) do not appear on the command line either. As such, argc does not count these, either.

Argv returns a pointer to a string (allocated on the heap) of a specified command line item. To use argv you simply load ax with a value between one and the number returned by argc and execute the argv routine. On return, es:di points at a string containing the specified command line option. If the number in ax is greater than the number of command line arguments, then argv returns a pointer to an empty string (i.e., a zero byte). Since argv calls malloc to allocate storage on the heap, there is the possibility that a memory allocation error will occur. Argv returns the carry set if a memory allocation error occurs. Remember to free the storage allocated to a command line parameter after you are through with it.

Example: The following code echoes the command line parameters to the screen.

```
                 include    stdlib.a
                 includelib stdlib.lib

dseg             segment  para public 'data'

ArgCnt           word     0

dseg             ends

cseg             segment    para public 'code'
                 assume     cs:cseg, ds:dseg

Main             proc
                 mov      ax, dseg
                 mov      ds, ax
                 mov      es, ax

; Must call the memory manager initialization routine if you use
; any routine which calls malloc!  ARGV is a good example of a
; routine which calls malloc.

                 meminit

                 argc                    ;Get the command line arg count.
                 jcxz     Quit           ;Quit if no cmd ln args.
                 mov      ArgCnt, 1      ;Init Cmd Ln count.
PrintCmds:       printf                  ;Print the item.
                 byte     "\n%2d: ",0
                 dword    ArgCnt

                 mov      ax, ArgCnt     ;Get the next command line guy.
                 argv
                 puts
                 inc      ArgCnt         ;Move on to next arg.
                 loop     PrintCmds      ;Repeat for each arg.
                 putcr

Quit:            ExitPgm                 ;DOS macro to quit program.
```

```
Main            endp
cseg            ends

sseg            segment    para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

;zzzzzzseg is required by the standard library routines.

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end        Main
```

13.4 UCR Standard Library File I/O Routines

Although MS-DOS' file I/O facilities are not too bad, the UCR Standard Library provides a file I/O package which makes blocked sequential I/O as easy as character at a time file I/O. Furthermore, with a tiny amount of effort, you can use all the StdLib routines like printf, print, puti, puth, putc, getc, gets, etc., when performing file I/O. This greatly simplifies text file operations in assembly language.

Note that record oriented, or binary I/O, is probably best left to pure DOS. any time you want to do random access within a file. The Standard Library routines really only support sequential text I/O. Nevertheless, this is the most common form of file I/O around, so the Standard Library routines are quite useful indeed.

The UCR Standard Library provides eight file I/O routines: fopen, fcreate, fclose, fgetc, fread, fputc, and fwrite. Fgetc and fputc perform character at a time I/O, fread and fwrite let you read and write blocks of data, the other four functions perform the obvious DOS operations.

The UCR Standard Library uses a special *file variable* to keep track of file operations. There is a special record type, *FileVar,* declared in stdlib.a[8]. When using the StdLib file I/O routines you must create a variable of type FileVar for every file you need open at the same time. This is very easy, just use a definition of the form:

```
MyFileVar   FileVar              {}
```

Please note that a Standard Library file variable *is not* the same thing as a DOS file handle. It is a structure which contains the DOS file handle, a buffer (for blocked I/O), and various index and status variables. The internal structure of this type is of no interest (remember data encapsulation!) except to the implementor of the file routines. You will pass the address of this file variable to the various Standard Library file I/O routines.

13.4.1 Fopen

Entry parameters: ax- File open mode
0- File opened for reading
1- File opened for writing
dx:si- Points at a zero terminated string containing the filename.
es:di- Points at a StdLib file variable.

Exit parameters: If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fopen opens a sequential text file for reading *or* writing. Unlike DOS, you cannot open a file for reading and writing. Furthermore, this is a sequential text file which does not support random access. Note that the file must exist or fopen will return an error. This is even true when you open the file for writing.

8. Actually, it's declared in *file.a*. Stdlib.a includes file.a so this definition appears inside stdlib.a as well.

Note that if you open a file for writing and that file already exists, any data written to the file will overwrite the existing data. When you close the file, any data appearing in the file after the data you wrote will still be there. If you want to erase the existing file before writing data to it, use the fcreate function.

13.4.2 Fcreate

Entry parameters: dx:si- Points at a zero terminated string containing the filename.
es:di- Points at a StdLib file variable.

Exit parameters: If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fcreate creates a new file and opens it for writing. If the file already exists, fcreate deletes the existing file and creates a new one. It initializes the file variable for output but is otherwise identical to the fopen call.

13.4.3 Fclose

Entry parameters: es:di- Points at a StdLib file variable.
Exit parameters: If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fclose closes a file and updates any internal housekeeping information. *It is very important that you close all files opened with fopen or fcreate using this call*. When making DOS file calls, if you forget to close a file DOS will automatically do that for you when your program terminates. However, the StdLib routines cache up data in internal buffers. the fclose call automatically flushes these buffers to disk. If you exit your program without calling fclose, you may lose some data written to the file but not yet transferred from the internal buffer to the disk.

If you are in an environment where it is possible for someone to abort the program without giving you a chance to close the file, you should call the fflush routines (see the next section) on a regular basis to avoid losing too much data.

13.4.4 Fflush

Entry parameters: es:di- Points at a StdLib file variable.
Exit parameters: If the carry is set, **ax** contains the returned DOS error code (see DOS open function).

This routine immediately writes any data in the internal file buffer to disk. Note that you should only use this routine in conjunction with files opened for writing (or opened by fcreate). If you write data to a file and then need to leave the file open, but inactive, for some time period, you should perform a flush operation in case the program terminates abnormally.

13.4.5 Fgetc

Entry parameters: es:di- Points at a StdLib file variable.
Exit parameters: If the carry flag is clear, al contains the character read from the file.
If the carry is set, ax contains the returned DOS error code (see DOS open function).
ax will contain zero if you attempt to read beyond the end of file.

Fgetc reads a single character from the file and returns this character in the al register. Unlike calls to DOS, single character I/O using fgetc is relatively fast since the StdLib routines use blocked I/O. Of course, multiple calls to fgetc will never be faster than a call to fread (see the next section), but the performance is not too bad.

Fgetc is very flexible. As you will see in a little bit, you may redirect the StdLib input routines to read their data from a file using fgetc. This lets you use the higher level routines like gets and getsm when reading data from a file.

13.4.6 Fread

Entry parameters:	es:di-	Points at a StdLib file variable.
	dx:si-	Points at an input data buffer.
	cx-	Contains a byte count.
Exit parameters:	If the carry flag is clear, ax contains the actual number of bytes read from the file.	
	If the carry is set, ax contains the returned DOS error code (see DOS open function).	

Fread is very similar to the DOS read command. It lets you read a block of bytes, rather than just one byte, from a file. Note that if all you are doing is reading a block of bytes from a file, the DOS call is slightly more efficient than fread. However, if you have a mixture of single byte reads and multi-byte reads, the combination of fread and fgetc work very well.

As with the DOS read operation, if the byte count returned in ax does not match the value passed in the cx register, then you've read the remaining bytes in the file. When this occurs, the next call to fread or fgetc will return an EOF error (carry will be set and ax will contain zero). Note that fread does not return EOF unless there were zero bytes read from the file.

13.4.7 Fputc

Entry parameters:	es:di-	Points at a StdLib file variable.
	al-	Contains the character to write to the file.
Exit parameters:	If the carry is set, ax contains the returned DOS error code (see DOS open function).	

Fputc writes a single character (in al) to the file specified by the file variable whose address is in es:di. This call simply adds the character in al to an internal buffer (part of the file variable) until the buffer is full. Whenever the buffer is filled or you call fflush (or close the file with fclose), the file I/O routines write the data to disk.

13.4.8 Fwrite

Entry parameters:	es:di-	Points at a StdLib file variable.
	dx:si-	Points at an output data buffer.
	cx-	Contains a byte count.
Exit parameters:	If the carry flag is clear, ax contains the actual number of bytes written to the file.	
	If the carry is set, ax contains the returned DOS error code (see DOS open function).	

Like fread, fwrite works on blocks of bytes. It lets you write a block of bytes to a file opened for writing with fopen or fcreate.

13.4.9 Redirecting I/O Through the StdLib File I/O Routines

The Standard Library provides very few file I/O routines. Fputc and fwrite are the only two output routines, for example. The "C" programming language standard library (on which the UCR Standard Library is based) provides many routines like *fprintf, fputs, fscanf,* etc. None of these are necessary in the UCR Standard Library because the UCR library provides an I/O redirection mechanism that lets you reuse all existing I/O routines to perform file I/O.

The UCR Standard Library putc routine consists of a single jmp instruction. This instruction transfers control to some actual output routine via an indirect address internal to the putc code. Normally, this pointer variable points at a piece of code which writes the character in the al register to the DOS standard output device. However, the Standard Library also provides four routines which let you manipulate this indirect pointer. By changing this pointer you can redirect the output from its current routine to a routine of your choosing. *All* Standard Library output routines (e.g., printf, puti, puth, puts) call putc to output individual characters. Therefore, redirecting the putc routine affects all the output routines.

Likewise, the getc routine is nothing more than an indirect jmp whose pointer variable normally points at a piece of code which reads data from the DOS standard input. Since all Standard Library input routines call the getc function to read each character you can redirect file input in a manner identical to file output.

The Standard Library *GetOutAdrs, SetOutAdrs, PushOutAdrs,* and *PopOutAdrs* are the four main routines which manipulate the output redirection pointer. GetOutAdrs returns the address of the current output routine in the es:di registers. Conversely, SetOutAdrs expects you to pass the address of a new output routine in the es:di registers and it stores this address into the output pointer. PushOutAdrs and PopOutAdrs push and pop the pointer on an internal stack. These do not use the 80x86's hardware stack. You are limited to a small number of pushes and pops. Generally, you shouldn't count on being able to push more than four of these addresses onto the internal stack without overflowing it.

GetInAdrs, SetInAdrs, PushInAdrs, and *PopInAdrs* are the complementary routines for the input vector. They let you manipulate the input routine pointer. Note that the stack for PushInAdrs/PopInAdrs is not the same as the stack for PushOutAdrs/PopOutAdrs. Pushes and pops to these two stacks are independent of one another.

Normally, the output pointer (which we will henceforth refer to as the *output hook*) points at the Standard Library routine *PutcStdOut*[9]. Therefore, you can return the output hook to its normal initialization state at any time by executing the statements[10]:

```
mov      di, seg SL_PutcStdOut
mov      es, di
mov      di, offset SL_PutcStdOut
SetOutAdrs
```

The PutcStdOut routine writes the character in the al register to the DOS standard output, which itself might be redirected to some file or device (using the ">" DOS redirection operator). If you want to make sure your output is going to the video display, you can always call the PutcBIOS routine which calls the BIOS directly to output a character[11]. You can force all Standard Library output to the *standard error device* using a code sequence like:

```
mov      di, seg SL_PutcBIOS
mov      es, di
mov      di, offset SL_PutcBIOS
SetOutAdrs
```

Generally, you would not simply blast the output hook by storing a pointer to your routine over the top of whatever pointer was there and then restoring the hook to PutcStdOut upon completion. Who knows if the hook was pointing at PutcStdOut in the first place? The best solution is to use the Standard Library PushOutAdrs and PopOutAdrs routines to preserve and restore the previous hook. The following code demonstrates a *gentler* way of modifying the output hook:

9. Actually, the routine is *SL_PutcStdOut*. The Standard Library macro by which you would normally call this routine is PutcStdOut.

10. If you do not have any calls to PutcStdOut in your program, you will also need to add the statement "externdef SL_PutcStdOut:far" to your program.

11. It is possible to redirect even the BIOS output, but this is rarely done and not easy to do from DOS.

```
                PushOutAdrs           ;Save current output routine.
                mov      di, seg Output_Routine
                mov      es, di
                mov      di, offset Output_Routine
                SetOutAdrs

       <Do all output to Output_Routine here>

                PopOutAdrs            ;Restore previous output routine.
```

Handle input in a similar fashion using the corresponding input hook access routines and the SL_GetcStdOut and SL_GetcBIOS routines. Always keep in mind that there are a limited number of entries on the input and output hook stacks so what how many items you push onto these stacks without popping anything off.

To redirect output to a file (or redirect input from a file) you must first write a short routine which writes (reads) a single character from (to) a file. This is very easy. The code for a subroutine to output data to a file described by file variable *OutputFile* is

```
ToOutput         proc     far
                 push     es
                 push     di

; Load ES:DI with the address of the OutputFile variable. This
; code assumes OutputFile is of type FileVar, not a pointer to
; a variable of type FileVar.

                 mov      di, seg OutputFile
                 mov      es, di
                 mov      di, offset OutputFile

; Output the character in AL to the file described by “OutputFile”

                 fputc

                 pop      di
                 pop      es
                 ret
ToOutput         endp
```

Now with only one additional piece of code, you can begin writing data to an output file using all the Standard Library output routines. That is a short piece of code which redirects the output hook to the “ToOutput” routine above:

```
SetOutFile       proc
                 push     es
                 push     di

                 PushOutAdrs                     ;Save current output hook.
                 mov      di, seg ToOutput
                 mov      es, di
                 mov      di, offset ToOutput
                 SetOutAdrs

                 pop      di
                 pop      es
                 ret
SetOutFile       endp
```

There is no need for a separate routine to restore the output hook to its previous value; PopOutAdrs will handle that task by itself.

13.4.10 A File I/O Example

The following piece of code puts everything together from the last several sections. This is a short program which adds line numbers to a text file. This program expects two command line parameters: an input file and an output file. It copies the input file to the output file while appending line numbers to the beginning of each line in the output file. This code demonstrates the use of argc, argv, the Standard Library file I/O routines, and I/O redirection.

```
; This program copies the input file to the output file and adds
; line numbers while it is copying the file.

                include     stdlib.a
                includelib  stdlib.lib

dseg            segment     para public 'data'

ArgCnt          word        0
LineNumber      word        0
DOSErrorCode    word        0
InFile          dword       ?                     ;Ptr to Input file name.
OutFile         dword       ?                     ;Ptr to Output file name
InputLine       byte        1024 dup (0)          ;Input/Output data buffer.
OutputFile      FileVar     {}
InputFile       FileVar     {}

dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

; ReadLn- Reads a line of text from the input file and stores the
;         data into the InputLine buffer:

ReadLn          proc
                push     ds
                push     es
                push     di
                push     si
                push     ax

                mov      si, dseg
                mov      ds, si
                mov      si, offset InputLine
                lesi     InputFile

GetLnLp:
                fgetc
                jc       RdLnDone                 ;If some bizzarre error.
                cmp      ah, 0                    ;Check for EOF.
                je       RdLnDone                 ;Note:carry is set.
                mov      ds:[si], al
                inc      si
                cmp      al, lf                   ;At EOLN?
                jne      GetLnLp
                dec      si                       ;Back up before LF.
                cmp      byte ptr ds:[si-1], cr ;CR before LF?
                jne      RdLnDone
                dec      si                       ;If so, skip it too.

RdLnDone:       mov      byte ptr ds:[si], 0 ;Zero terminate.
                pop      ax
                pop      si
                pop      di
                pop      es
                pop      ds
                ret
ReadLn          endp

; MyOutput- Writes the single character in AL to the output file.

MyOutput        proc     far
                push     es
                push     di
                lesi     OutputFile
                fputc
                pop      di
                pop      es
                ret
MyOutput        endp

; The main program which does all the work:

Main            proc
```

```
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Must call the memory manager initialization routine if you use
; any routine which calls malloc!  ARGV is a good example of a
; routine calls malloc.

                meminit

; We expect this program to be called as follows:
;               fileio file1, file2
; anything else is an error.

                argc
                cmp     cx, 2           ;Must have two parameters.
                je      Got2Parms
BadParms:       print
                byte    "Usage: FILEIO infile, outfile",cr,lf,0
                jmp     Quit

; Okay, we've got two parameters, hopefully they're valid names.
; Get copies of the filenames and store away the pointers to them.

Got2Parms:      mov     ax, 1           ;Get the input filename
                argv
                mov     word ptr InFile, di
                mov     word ptr InFile+2, es

                mov     ax, 2           ;Get the output filename
                argv
                mov     word ptr OutFile, di
                mov     word ptr OutFile+2, es

; Output the filenames to the standard output device

                printf
                byte    "Input file: %^s\n"
                byte    "Output file: %^s\n",0
                dword   InFile, OutFile

; Open the input file:

                lesi    InputFile
                mov     dx, word ptr InFile+2
                mov     si, word ptr InFile
                mov     ax, 0
                fopen
                jnc     GoodOpen
                mov     DOSErrorCode, ax
                printf
                byte    "Could not open input file, DOS: %d\n",0
                dword   DOSErrorCode
                jmp     Quit

; Create a new file for output:

GoodOpen:       lesi    OutputFile
                mov     dx, word ptr OutFile+2
                mov     si, word ptr OutFile
                fcreate
                jnc     GoodCreate
                mov     DOSErrorCode, AX
                printf
                byte    "Could not open output file, DOS: %d\n",0
                dword   DOSErrorCode
                jmp     Quit

; Okay, save the output hook and redirect the output.

GoodCreate:     PushOutAdrs
                lesi    MyOutput
                SetOutAdrs

WhlNotEOF:      inc     LineNumber

; Okay, read the input line from the user:
```

```
                call     ReadLn
                jc       BadInput

; Okay, redirect the output to our output file and write the last
; line read prefixed with a line number:

                printf
                byte     "%4d:   %s\n",0
                dword   LineNumber, InputLine
                jmp      WhlNotEOF

BadInput:       push     ax           ;Save error code.
                PopOutAdrs            ;Restore output hook.
                pop      ax           ;Retrieve error code.
                test     ax, ax       ;EOF error? (AX = 0)
                jz       CloseFiles
                mov     DOSErrorCode, ax
                printf
                byte     "Input error, DOS: %d\n",0
                dword    LineNumber

; Okay, close the files and quit:

CloseFiles:     lesi    OutputFile
                fclose
                lesi     InputFile
                fclose

Quit:           ExitPgm               ;DOS macro to quit program.
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end        Main
```

13.5 Sample Program

If you want to use the Standard Library's output routines (putc, print, printf, etc.) to output data to a file, you can do so by manually redirecting the output before and after each call to these routines. Unfortunately, this can be a lot of work if you mix interactive I/O with file I/O. The following program presents several macros that simplify this task for you.

```
; FileMacs.asm
;
; This program presents a set of macros that make file I/O with the
; Standard Library even easier to do.
;
; The main program writes a multiplication table to the file "MyFile.txt".

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment  para public 'data'

CurOutput       dword    ?

Filename        byte     "MyFile.txt",0

i               word     ?
j               word     ?
```

```
TheFile         filevar  {}

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg


; For-Next macros from Chapter Eight.
; See Chapter Eight for details on how this works.

ForLp           macro    LCV, Start, Stop
                local    ForLoop

                ifndef   $$For&LCV&
$$For&LCV&=     0
                else
$$For&LCV&=     $$For&LCV& + 1
                endif

                mov      ax, Start
                mov      LCV, ax

ForLoop         textequ  @catstr($$For&LCV&, %$$For&LCV&)
&ForLoop&:
                mov      ax, LCV
                cmp      ax, Stop
                jg       @catstr($$Next&LCV&, %$$For&LCV&)
                endm



Next            macro    LCV
                local    NextLbl
                inc      LCV
                jmp      @catstr($$For&LCV&, %$$For&LCV&)
NextLbl         textequ  @catstr($$Next&LCV&, %$$For&LCV&)
&NextLbl&:
                endm


; File I/O macros:
;
;
; SetPtr sets up the CurOutput pointer variable.  This macro is called
; by the other macros, it's not something you would normally call directly.
; Its whole purpose in life is to shorten the other macros and save a little
; typing.

SetPtr          macro    fvar
                push     es
                push     di

                mov      di, offset fvar
                mov      word ptr CurOutput, di
                mov      di, seg fvar
                mov      word ptr CurOutput+2, di

                PushOutAdrs
                lesi     FileOutput
                SetOutAdrs
                pop      di
                pop      es
                endm
;
;
;
; fprint-         Prints a string to the display.
```

```
;
; Usage:
;                 fprint   filevar,"String or bytes to print"
;
; Note: you can supply optional byte or string data after the string above by
;      enclosing the data in angle brackets, e.g.,
;
;                 fprint   filevar,<"string to print",cr,lf>
;
; Do *NOT* put a zero terminating byte at the end of the string, the fprint
; macro will do that for you automatically.

fprint           macro    fvar:req, string:req
                 SetPtr   fvar

                 print
                 byte     string
                 byte     0

                 PopOutAdrs
                 endm

; fprintf-       Prints a formatted string to the display.
; fprintff-      Like fprintf, but handles floats as well as other items.
;
; Usage:
;                fprintf  filevar,"format string", optional data values
;                fprintff filevar,"format string", optional data values
; Examples:
;
;      fprintf   FileVariable,"i=%d, j=%d\n", i, j
;      fprintff  FileVariable,"f=%8.2f, i=%d\n", f, i
;
; Note: if you want to specify a list of strings and bytes for the format
;       string, just surround the items with an angle bracket, e.g.,
;
;      fprintf FileVariable, <"i=%d, j=%d",cr,lf>, i, j
;
;

fprintf          macro    fvar:req, FmtStr:req, Operands:vararg
                 setptr   fvar

                 printf
                 byte     FmtStr
                 byte     0

                 for      ThisVal, <Operands>
                 dword    ThisVal
                 endm

                 PopOutAdrs
                 endm

fprintff         macro    fvar:req, FmtStr:req, Operands:vararg
                 setptr   fvar

                 printff
                 byte     FmtStr
                 byte     0

                 for      ThisVal, <Operands>
                 dword    ThisVal
                 endm

                 PopOutAdrs
                 endm


; F-   This is a generic macro that converts stand-alone (no code stream parms)
```

```
;       stdlib functions into file output routines.  Use it with putc, puts,
;       puti, putu, putl, putisize, putusize, putlsize, putcr, etc.
;
; Usage:
;
;       F          StdLibFunction, FileVariable
;
; Examples:
;
;       mov        al, 'A'
;       F          putc, TheFile
;       mov        ax, I
;       mov        cx, 4
;       F          putisize, TheFile


F                  macro     func:req, fvar:req
                   setptr    fvar
                   func
                   PopOutAdrs
                   endm

; WriteLn- Quick macro to handle the putcr operation (since this code calls
; putcr so often).

WriteLn            macro     fvar:req
                   F         putcr, fvar
                   endm


; FileOutput- Writes the single character in AL to an output file.
; The macros above redirect the standard output to this routine
; to print data to a file.

FileOutput         proc      far
                   push      es
                   push      di
                   push      ds
                   mov       di, dseg
                   mov       ds, di

                   les       di, CurOutput
                   fputc

                   pop       ds
                   pop       di
                   pop       es
                   ret
FileOutput         endp


; A simple main program that tests the code above.
; This program writes a multiplication table to the file "MyFile.txt"

Main               proc
                   mov       ax, dseg
                   mov       ds, ax
                   mov       es, ax
                   meminit

; Rewrite(TheFile, FileName);

                   ldxi      FileName
                   lesi      TheFile
                   fcreate

; writeln(TheFile);
; writeln(TheFile,'    ');
; for i := 0 to 5 do write(TheFile,'|',i:4,' ');
; writeln(TheFile);
```

```
                WriteLn   TheFile
                fprint    TheFile,"    "

                forlp     i,0,5
                fprintf   TheFile, "|%4d ", i
                next      i
                WriteLn   TheFile

; for j := -5 to 5 do begin
;
;      write(TheFile,'----');
;      for i := 0 to 5 do write(TheFile, '+-----');
;      writeln(TheFile);
;
;      write(j:3, ' |');
;      for i := 0 to 5 do write(i*j:4, ' |);
;      writeln(TheFile);
;
; end;

                forlp     j,-5,5

                fprint    TheFile,"----"
                forlp     i,0,5
                fprintf   TheFile,"+-----"
                next      i
                fprint    TheFile,<"+",cr,lf>

                fprintf   TheFile, "%3d |", j

                forlp     i,0,5

                mov       ax, i
                imul      j
                mov       cx, 4
                F         putisize, TheFile
                fprint    TheFile, " |"

                next      i
                Writeln   TheFile

                next      j
                WriteLn   TheFile

; Close(TheFile);

                lesi      TheFile
                fclose


Quit:           ExitPgm                ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment   para stack 'stack'
stk             db        1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       db        16 dup (?)
zzzzzzseg       ends
                end       Main
```

13.6 Laboratory Exercises

The following three programs all do the same thing: they copy the file ”ex13_1.in” to the file “ex13_1.out”. The difference is the way they copy the files. The first program, ex13_1a, copies the data from the input file to the output file using character at a time I/O under DOS. The second program, ex13_1b, uses blocked I/O under DOS. The third program, ex13_1c, uses the Standard Library’s file I/O routines to copy the data.

Run these three programs and measure the amount of time they take to run[12]. **For your lab report:** report the running times and comment on the relative efficiencies of these data transfer methods. Is the loss of performance of the Standard Library routines (compared to block I/O) justified in terms of the ease of use of these routines? Explain.

```
; EX13_1a.asm
;
; This program copies one file to another using character at a time I/O.
; It is easy to write, read, and understand, but character at a time I/O
; is quite slow.  Run this program and time its execution.  Then run the
; corresponding blocked I/O exercise and compare the execution times of
; the two programs.

                 include    stdlib.a
                 includelib stdlib.lib


dseg             segment  para public 'data'

FHndl            word     ?
FHndl2           word     ?
Buffer           byte     ?

FName            equ      this word
FNamePtr         dword    FileName

Filename         byte     "Ex13_1.in",0
Filename2        byte     "Ex13_1.out",0

dseg             ends


cseg             segment  para public 'code'
                 assume   cs:cseg, ds:dseg

Main             proc
                 mov      ax, dseg
                 mov      ds, ax
                 mov      es, ax
                 meminit

                 mov      ah, 3dh       ;Open the input file
                 mov      al, 0         ; for reading
                 lea      dx, Filename  ;DS points at filename’s
                 int      21h           ; segment
                 jc       BadOpen
                 mov      FHndl, ax     ;Save file handle


                 mov      FName, offset Filename2 ;Set this up in case there
                 mov      FName+2, seg FileName2  ; is an error during open.

                 mov      ah, 3ch       ;Open the output file for writing
                 mov      cx, 0         ; with normal file attributes
```

12. If you have a really fast machine you may want to make the ex13_1.in file larger (by copying and pasting data in the file) to make it larger.

```
                lea     dx, Filename2 ;Presume DS points at filename
                int     21h           ; segment
                jc      BadOpen
                mov     FHndl2, ax    ;Save file handle

LP:             mov     ah,3fh        ;Read data from the file
                lea     dx, Buffer    ;Address of data buffer
                mov     cx, 1         ;Read one byte
                mov     bx, FHndl     ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx        ;EOF reached?
                jne     EOF

                mov     ah,40h        ;Write data to the file
                lea     dx, Buffer    ;Address of data buffer
                mov     cx, 1         ;Write one byte
                mov     bx, FHndl2    ;Get file handle value
                int     21h
                jc      WriteError
                jmp     LP            ;Read next byte

EOF:            mov     bx, FHndl
                mov     ah, 3eh       ;Close file
                int     21h
                jmp     Quit

ReadError:      printf
                byte    "Error while reading data from file '%s'.",cr,lf,0
                dword   FileName
                jmp     Quit

WriteError:     printf
                byte    "Error while writing data to file '%s'.",cr,lf,0
                dword   FileName2
                jmp     Quit

BadOpen:        printf
                byte    "Could not open '%^s'.  Make sure this file is “
                byte    “in the ",cr,lf
                byte    "current directory before attempting to run “
                byte    this program again.", cr,lf,0
                dword   FName

Quit:           ExitPgm              ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

```
; EX13_1b.asm
;
; This program copies one file to another using blocked I/O.
; Run this program and time its execution.  Compare the execution time of
; this program against that of the character at a time I/O and the
; Standard Library File I/O example (ex13_1a and ex13_1c).

                include   stdlib.a
```

```
                includelib  stdlib.lib

dseg            segment  para public 'data'

; File handles for the files we will open.

FHndl           word     ?                      ;Input file handle
FHndl2          word     ?                      ;Output file handle

Buffer          byte     256 dup (?)            ;File buffer area

FName           equ      this word              ;Ptr to current file name
FNamePtr        dword    FileName

Filename        byte     "Ex13_1.in",0          ;Input file name
Filename2       byte     "Ex13_1.out",0         ;Output file name

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                mov      ah, 3dh                ;Open the input file
                mov      al, 0                  ; for reading
                lea      dx, Filename           ;Presume DS points at
                int      21h                    ; filename's segment
                jc       BadOpen
                mov      FHndl, ax              ;Save file handle


                mov      FName, offset Filename2 ;Set this up in case there
                mov      FName+2, seg FileName2  ; is an error during open.

                mov      ah, 3ch         ;Open the output file for writing
                mov      cx, 0           ; with normal file attributes
                lea      dx, Filename2 ;Presume DS points at filename
                int      21h             ; segment
                jc       BadOpen
                mov      FHndl2, ax      ;Save file handle


; The following loop reads 256 bytes at a time from the file and then
; writes those 256 bytes to the output file.

LP:             mov      ah,3fh         ;Read data from the file
                lea      dx, Buffer     ;Address of data buffer
                mov      cx, 256        ;Read 256 bytes
                mov      bx, FHndl      ;Get file handle value
                int      21h
                jc       ReadError
                cmp      ax, cx         ;EOF reached?
                jne      EOF

                mov      ah, 40h        ;Write data to file
                lea      dx, Buffer     ;Address of output buffer
                mov      cx, 256        ;Write 256 bytes
                mov      bx, FHndl2     ;Output handle
                int      21h
                jc       WriteError
                jmp      LP             ;Read next block

; Note, just because the number of bytes read does not equal 256,
```

```
; don't get the idea we're through, there could be up to 255 bytes
; in the buffer still waiting to be processed.

EOF:            mov     cx, ax          ;Put # of bytes to write in CX.
                jcxz    EOF2            ;If CX is zero, we're really done.
                mov     ah, 40h         ;Write data to file
                lea     dx, Buffer      ;Address of output buffer
                mov     bx, FHndl2      ;Output handle
                int     21h
                jc      WriteError


EOF2:           mov     bx, FHndl
                mov     ah, 3eh         ;Close file
                int     21h
                jmp     Quit

ReadError:      printf
                byte    "Error while reading data from file '%s'.",cr,lf,0
                dword   FileName
                jmp     Quit

WriteError:     printf
                byte    "Error while writing data to file '%s'.",cr,lf,0
                dword   FileName2
                jmp     Quit

BadOpen:        printf
                byte    "Could not open '%^s'.  Make sure this file is in “
                byte    “the ",cr,lf
                byte    "current directory before attempting to run “
                byte    “this program again.", cr,lf,0
                dword   FName

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg           ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main

---------------------------------------------------

; EX13_1c.asm
;
; This program copies one file to another using the standard library
; file I/O routines.  The Standard Library file I/O routines let you do
; character at a time I/O, but they block up the data to transfer to improve
; system performance.  You should find that the execution time of this
; code is somewhere between blocked I/O (ex13_1b) and character at a time
; I/O (EX13_1a);  it will, however, be much closer to the block I/O time
; (probably about twice as long as block I/O).

                include    stdlib.a
                includelib stdlib.lib


dseg            segment para public 'data'

InFile          filevar {}
OutFile         filevar {}

Filename        byte    "Ex13_1.in",0;Input file name
```

```
Filename2       byte     "Ex13_1.out",0;Output file name

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

; Open the input file:

                mov      ax, 0                   ;Open for reading
                ldxi     Filename
                lesi     InFile
                fopen
                jc       BadOpen

; Open the output file:

                mov      ax, 1                   ;Open for output
                ldxi     Filename2
                lesi     OutFile
                fcreate
                jc       BadCreate

; Copy the input file to the output file:

CopyLp:         lesi     InFile
                fgetc
                jc       GetDone

                lesi     OutFile
                fputc
                jmp      CopyLp

BadOpen:        printf
                byte     "Error opening '%s'",cr,lf,0
                dword    Filename
                jmp      Quit

BadCreate:      printf
                byte     "Error creating '%s'",cr,lf,0
                dword    Filename2
                jmp      CloseIn

GetDone:        cmp      ax, 0                   ;Check for EOF
                je       AtEOF

                print
                byte     "Error copying files (read error)",cr,lf,0

AtEOF:          lesi     OutFile
                fclose
CloseIn:        lesi     InFile
                fclose

Quit:           ExitPgm                          ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends
```

```
zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

13.7 Programming Projects

1) The sample program in Section 13.5 reroutes the standard output through the Standard Library's file I/O routines allowing you to use any of the output routines to write data to a file. Write a similar set of routines and macros that let you read data from a file using the Standard Library's input routines (getc, gets, getsm scanf, etc.). Redirect the input through the Standard Library's file input functions.

2) The last sample program in section 13.3.12 (copyuc.asm on the companion CD-ROM) copies one file to another, possibly converting lower case characters to upper case. This program currently parses the command line directly and uses blocked I/O to copy the data in the file. Rewrite this program using argv/argc to process the command line parameters and use the Standard Library file I/O routines to process each character in the file.

3) Write a "word count" program that counts the number of characters, words, and lines within a file. Assume that a word is any sequence of characters between spaces, tabs, carriage returns, line feeds, the beginning of a file, and the end of a file (if you want to save some effort, you can assume a "whitespace" symbol is any ASCII code less than or equal to a space).

4) Write a program that prints an ASCII text file to the printer. Use the BIOS int 17h services to print the characters in the file.

5) Write two programs, "xmit" and "rcv". The xmit program should fetch a command line filename and transmit this file across the serial port. It should transmit the filename and the number of bytes in the file (hint: use the DOS seek command to determine the length of the file). The rcv program should read the filename and file length from the serial port, create the file by the specified name, read the specified number of bytes from the serial port, and then close the file.

13.8 Summary

MS-DOS and BIOS provide many system services which control the hardware on a PC. They provide a machine independent and flexible interface. Unfortunately, the PC has grown up quite a bit since the days of the original 5 Mhz 8088 IBM PC. Many BIOS and DOS calls are now obsolete, having been superseded by newer calls. To ensure backwards compatibility, MS-DOS and BIOS generally support all of the older obsolete calls as well as the newer calls. However, your programs should not use the obsolete calls, they are there for backwards compatibility only.

The BIOS provides many services related to the control of devices such as the video display, the printer port, the keyboard, the serial port, the real time clock, etc. Descriptions of the BIOS services for these devices appear in the following sections:

- "INT 5- Print Screen" on page 702
- "INT 10h - Video Services" on page 702
- "INT 11h - Equipment Installed" on page 704
- "INT 12h - Memory Available" on page 704
- "INT 13h - Low Level Disk Services" on page 704
- "INT 14h - Serial I/O" on page 706
- "INT 15h - Miscellaneous Services" on page 708
- "INT 16h - Keyboard Services" on page 708
- "INT 17h - Printer Services" on page 710
- "INT 18h - Run BASIC" on page 712
- "INT 19h - Reboot Computer" on page 712

MS-DOS provides several different types of services. This chapter concentrated on the file I/O services provided by MS-DOS. In particular, this chapter dealt with implementing efficient file I/O operations using blocked I/O. To learn how to perform file I/O and perform other MS-DOS operations, check out the following sections:

Accessing command line parameters is an important operation within MS-DOS applications. DOS’ PSP (Program Segment Prefix) contains the command line and several other pieces of important information. To learn about the various fields in the PSP and see how to access command line parameters, check out the following sections in this chapter:

Of course, the UCR Standard Library provides some file I/O routines as well. This chapter closes up by describing some of the StdLib file I/O routines along with their advantages and disadvantages. See

13.9 Questions

1) How are BIOS routines called?

2) Which BIOS routine is used to write a character to the:

   a) video display b) serial port c) printer port

3) When the serial transmit or receive services return to the caller, the error status is returned in the AH register. However, there is a problem with the value returned. What is this problem?

4) Explain how you could test the keyboard to see if a key is available. 5)What is wrong with the keyboard shift status function?

6) How are special key codes (those keystrokes not returning ASCII codes) returned by the read keyboard call?

7) How would you send a character to the printer?

8) How do you read the real time clock?

9) Given that the RTC increments a 32-bit counter every 55ms, how long will the system run before overflow of this counter occurs?

10) Why should you reset the clock if, when reading the clock, you've determined that the counter has overflowed?

11) How do assembly language programs call MS-DOS?

12) Where are parameters generally passed to MS-DOS?

13) Why are there two sets of filing functions in MS-DOS?

14) Where can the DOS command line be found?

15) What is the purpose of the environment string area?

16) How can you determine the amount of memory available for use by your program?

17) Which is more efficient: character I/O or blocked I/O? Why?

18) What is a good blocksize for blocked I/O?

19) What can't you use blocked I/O on random access files?

20) Explain how to use the seek command to move the file pointer 128 bytes backwards in the file from the current file position.

21) Where is the error status normally returned after a call to DOS?

22) Why is it difficult to use blocked I/O on a random access file? Which would be easier, random access on a blocked I/O file opened for input or random access on a blocked I/O file opened for reading and writing?

23) Describe how you might implement blocked I/O on files opened for random access reading and writing.

24) What are two ways you can obtain the address of the PSP?

25) How do you determine that you've reached the end of file when using MS-DOS file I/O calls? When using UCR Standard Library file I/O calls?