Microsoft® MS™-DOS

Operating System

Programmer's Reference Manual

Microsoft Corporation

Comments about this documentation may be sent to:

Microsoft Corporation
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

Document No. 8411-310-02
Part No. 036-014-012

System Requirements

Disk drive(s)

One disk drive if and only if output is sent to the same physical disk from which the input was taken. None of the programs allows time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

For more information about other Microsoft products, contact:

Microsoft Corporation
10700 Northup Way
Bellevue, WA 98004
(206) 828-8080

Contents

Chapter 1
System Calls

CHAPTER 1

SYSTEM CALLS

1.1 INTRODUCTION

The routines that MS-DOS uses to manage system operation and resources can be called by any application program. Using these system calls makes it easier to write machine-independent programs and increases the likelihood that a program will be compatible with future versions of MS-DOS. MS-DOS system calls fall into several categories:

Standard character device I/O

Memory management

Process management

File and directory management

Microsoft Network calls

Miscellaneous system functions

MS-DOS services are invoked by an application by software interrupts. The current range of interrupts used for MS-DOS is 20H-27H, with 28H-40H reserved. Interrupt 21H is the function request service, and provides access to a wide variety of MS-DOS services. The selection of the Interrupt 21H function is through a function number placed in the AH register by the application. In some cases, the full AX register is used to specify the requested function. Each interrupt or function request uses values in various registers to receive or return function-specific information.

1.1.1 System Calls That Have Been Superseded

Many system calls introduced in versions of MS-DOS earlier than 2.0 have been superseded by function requests that are simpler to use and make better use of system resources. Although MS-DOS still includes these old system calls, they should not be used unless it is imperative that a program maintain backward-compatibility with the pre-2.0 versions of MS-DOS.

A table of the pre-2.0 system calls and a description of the File Control Block (required by some of the old calls) appears in Section 1.8, "Old System Calls."

The first part of this chapter explains how DOS manages its resources -- such as memory, files, and processes -- and briefly describes the purpose of most of the system calls. The remainder of the chapter describes each interrupt and function request in detail. The system call descriptions are in numeric order, interrupts followed by function requests. These descriptions include further detail on how MS-DOS manages its resources.

Chapter 2 of this book describes how to write an MS-DOS device driver. Chapters 3, 4, and 5 contain more detailed information about MS-DOS, including how it manages disk space, the control blocks it uses, and how it loads and executes relocatable programs (files with an extension of .EXE). Chapter 6 describes the Intel(R) object module format. Chapter 7 gives some programming hints.

1.2 STANDARD CHARACTER DEVICE I/O

The standard character function requests handle all input and output to and from character devices such as the console, printer, and serial ports. If a program uses these function requests, its input and output can be redirected.

Table 1.1 lists the MS-DOS function requests for managing standard character input and output.

Table 1.1 Standard Character I/O Function Requests

01H	Read Keyboard and Echo	Gets a character from standard input and echoes it to standard output.
02H	Display Character	Sends a character to standard output.
03H	Auxiliary Input	Gets a character from standard auxiliary.
04H	Auxiliary Output	Sends a character to standard auxiliary.
05H	Print Character	Sends a character to the standard printer.
06H	Direct Console I/O	Gets a character from standard input or sends a character to standard output.
07H	Direct Console Input	Gets a character from standard input.
08H	Read Keyboard	Gets a character from standard input.
09H	Display String	Sends a string to standard output.
0AH	Buffered Keyboard Input	Gets a string from standard input.
0BH	Check Keyboard Status	Reports on the status of the standard input buffer.
0CH	Flush Buffer, Read Keyboard	Empties the standard input buffer and calls one of the other standard character I/O function requests.

Although several of these standard character I/O function requests seem to do the same thing, they are distinguished by whether they echo characters from standard input to standard output or check for control characters. The detailed descriptions later in this chapter point out the differences.

1.3 MEMORY MANAGEMENT

MS-DOS keeps track of which areas of memory are allocated by writing a memory control block at the beginning of each area of memory. This control block specifies the size of the memory area; the name of the process, if any, that owns the memory area; and a pointer to the next area of memory. If the memory area is not owned, it is available.

Table 1.2 lists the MS-DOS function requests for managing memory.

Table 1.2 Memory Management Function Requests

48H	Allocate Memory	Requests a block of memory.
49H	Free Allocated Memory	Frees a block of memory previously allocated with 48H.
4AH	Set Block	Changes the size of an allocated memory block.

When a process requests additional memory with Function 48H, MS-DOS searches for a block of available memory large enough to satisfy the request. If it finds such a block of memory, it changes the memory control block to show the owning process. If the block of memory is larger than the requested amount, MS-DOS changes the size field of the memory control block to the requested amount, writes a new memory control block at the beginning of the unneeded portion that shows it is available, and updates the pointers to add this memory to the chain of memory control blocks. MS-DOS then returns the segment address of the first byte of the allocated memory to the requesting process.

When a process releases an allocated block of memory with Function 49H, DOS changes the memory control block to show that it is available (not owned by any process).

When a process shrinks an allocated block of memory with Function 4AH, DOS builds a memory control block for the memory being released and adds it to the chain of memory control blocks. When a process tries to expand an allocated block of memory with Function 4AH, MS-DOS treats it as a request for additional memory; rather than returning the segment address of the additional memory to the requesting process, however, MS-DOS simply chains the additional memory to the existing memory block.

If MS-DOS can't find a block of available memory large enough to satisfy a request for additional memory -- made with either Function 48H or Function 4AH -- MS-DOS returns an error code to the requesting process.

When a program receives control, it should call Function 4AH to shrink its initial memory allocation block (the block that begins with its Program Segment Prefix) to the minimum it requires. This frees unneeded memory and makes the best application design for portability to future multitasking environments.

When a program exits, MS-DOS automatically frees its initial memory allocation block before returning control to the calling program (COMMAND.COM is usually the calling program for application programs). The DOS frees any memory owned by the process exiting.

Any program that changes memory not allocated to it will most likely destroy at least one memory management control block. This causes a memory allocation error the next time MS-DOS tries to use the chain of memory control blocks; the only cure is to restart the system.

1.4 PROCESS MANAGEMENT

MS-DOS uses several function requests to load, execute, and terminate programs. Application programs can use these same function requests to manage other programs.

Table 1.3 lists the MS-DOS function requests for managing processes.

Table 1.3 Process Management Function Requests

31H	Keep Process	Terminates a process and returns control to the invoking process, but keeps the terminated process in memory.
4B00H	Load and Execute Program	Loads and executes a program.
4B03H	Load Overlay	Loads a program overlay without executing it.
4CH	End Process	Returns control to the invoking process.
4DH	Get Return Code of Child Process	Returns a code passed by a child process when it exits.
62H	Get PSP	Returns the segment address of the Program Segment Prefix of the current process.

1.4.1 Loading And Executing A Program

When a program loads and executes another program with Function 4B00H, MS-DOS allocates memory, writes a Program Segment Prefix (PSP) for the new program at offset 0 of the allocated memory, loads the new program, and passes control to it. When the invoked program exits, control returns to the calling program.

COMMAND.COM uses Function 4B00H to load and execute command files. Application programs have the same degree of control over process management as COMMAND.COM.

In addition to these common features, there are some differences in the way MS-DOS loads .COM and .EXE files.

Loading a .COM Program

When COMMAND.COM loads and executes a .COM program, it allocates all of available memory to the application and sets the stack pointer 100H bytes from the end of available memory. A .COM program should set up its own stack before shrinking its initial memory allocation block with Function 4AH, because the default stack is in the memory to be released.

If a newly loaded program is allocated all of memory -- as a .COM program is -- or requests all of available memory with Function 48H, MS-DOS allocated to it the memory occupied by the transient part of COMMAND.COM. If the program changes this memory, MS-DOS must reload the transient portion of COMMAND.COM before it can continue. If a program exits (via call 31H, Keep Process) without releasing enough memory, the system halts and must be reset. To minimize this possibility, a .COM program should shrink its initial allocation block with Function 4AH before doing anything else, and all programs must release all memory they allocate with Function 48H before exiting.

Loading an .EXE Program

When COMMAND.COM loads and executes an .EXE program, it allocates the size of the program's memory image plus either the value in the MAXALLOC field (offset 0CH) of the file header, if that much memory is available, or the value in the MINALLOC field (offset 0AH). These fields are set by the linker. Before passing control to the .EXE file, MS-DOS calculates the correct relocation addresses, based on the relocation information in the file header.

For a more detailed description of how MS-DOS loads .COM and .EXE files, see Chapters 3 and 4.

Executing a Program From Within Another Program

Because COMMAND.COM takes care of details such as building complete pathnames, searching the directory path for executable files, and relocating .EXE files, the simplest way to load and execute a program is to load and execute an additional copy of COMMAND.COM, passing it a command line that includes the /C switch to invoke the .COM or .EXE file. The description of Function 4B00H (Load and Execute Program) describes how to do this.

1.4.2 Loading An Overlay

When a program loads an overlay with Function 4B03H, it must pass to MS-DOS the segment address at which the overlay is to be loaded. The program then must call the overlay, and the overlay returns directly to the calling program. The calling program is in complete control: MS-DOS does not write a PSP for the overlay or intervene in any other way.

MS-DOS does not check to see if the calling program owns the memory where the overlay is to be loaded. If the calling program does not own the memory, loading the overlay will most likely destroy a memory control block, causing an eventual memory allocation error.

A program that loads an overlay must, therefore, either allow room for the overlay when it calls Function 4AH to shrink its initial memory allocation block, or should shrink its initial memory allocation block to the minimum and then use Function 48H to allocate memory for the overlay.

1.5 FILE AND DIRECTORY MANAGEMENT

The MS-DOS hierarchical (multilevel) file system is similar to that of the XENIX operating system. For a description of the multilevel directory system and how to use it, see the <u>MS-DOS User's Reference</u>.

1.5.1 Handles

To create or open a file, a program passes to MS-DOS a pathname and the attribute to be assigned to the file. MS-DOS returns a 16-bit number called a handle. For most subsequent actions, MS-DOS requires only this handle to identify the file.

A handle can refer to either a file or a device. MS-DOS predefines five standard handles. These handles are always open; you needn't open them before you use them. Table 1.4 lists these predefined handles.

Table 1.4 Predefined Device Handles

Handle	Standard device	Comment
0	Input	Can be redirected from command line
1	Output	Can be redirected from command line
2	Error	
3	Auxiliary	
4	Printer	

When MS-DOS creates or opens a file, it assigns the first available handle. A program can have 20 open handles; this includes the five predefined handles, so a program can typically open 15 extra files. Any of the five predefined handles can be temporarily forced to refer to an alternate file or device using function request 46H.

1.5.2 File-Related Function Requests

MS-DOS treats a file as a string of bytes; it assumes no record structure or access technique. An application program imposes whatever record structure it needs on this string of bytes. Reading from or writing to a file requires only pointing to the data buffer and specifying the number of bytes to read or write.

Table 1.5 lists the MS-DOS function requests for managing files.

Table 1.5 File-Related Function Requests

3CH	Create Handle	Creates a file.
3DH	Open Handle	Opens a file.
3EH	Close Handle	Closes a file.
3FH	Read Handle	Reads from a file.
40H	Write Handle	Writes to a file.
42H	Move File Pointer	Sets the read/write pointer in a file.
45H	Duplicate File Handle	Creates a new handle that refers to the same file as an existing handle.
46H	Force Duplicate File Handle	Makes an existing handle refer to the same file as another existing handle.
5AH	Create Temporary File	Creates a file with a unique name.
5BH	Create New File	Attempts to create a file, but fails if a file with the same name exists.

File Sharing

Version 3.1 of MS-DOS introduces file sharing, which lets more than one process share access to a file. File sharing operates only after the Share command has been executed to load file-sharing support. Table 1.6 lists the MS-DOS function requests for sharing files; if file sharing is not in effect, these function requests cannot be used. Function 3DH, Open Handle, can operate in several modes. Compatibility mode is usable without file sharing in effect. Here it is referred to in the file-sharing modes, which require file sharing to be in effect.

Table 1.6 File-Sharing Function Requests

3DH	Open Handle	Opens a file with one of the file-sharing modes.
440BH	IOCTL Retry	Specifies how many times an I/O operation that fails due to a file-sharing violation should be retried before Interrupt 24 is issued.
5C00H	Lock	Locks a region of a file.
5C01H	Unlock	Unlocks a region of a file.

1.5.3 Device-Related Function Requests

I/O Control for Devices is implemented with Function 44H (IOCTL); it includes several action codes to perform different device-related tasks. Some forms of the IOCTL function request require that the device driver be written to support the IOCTL interface. Table 1.7 lists the MS-DOS function requests for managing devices.

Table 1.7 Device-Related Function Requests

4400H,01H	IOCTL Data	Gets or sets device description.
4402H,03H	IOCTL Character	Gets or sets character device control data.
4404H,05H	IOCTL Block	Gets or sets block device control data.
4406H,07H	IOCTL Status	Checks device input or output status.
4408H	IOCTL Is Changeable	Checks whether block device contains removable medium.

Some forms of the IOCTL function request can only be used with Microsoft(R) Networks; they are listed in Section 1.6, "Microsoft Networks."

1.5.4 Directory-Related Function Requests

The root directory on a disk has room for a fixed number of entries: 64 on a standard single-sided disk, 112 on a standard double-sided disk. For hard disks, the number of directories is dependent on the DOS partition size. A subdirectory is simply a file with a unique attribute; there can be as many subdirectories on a disk as space allows. The depth of a directory structure, therefore, is limited only by the amount of storage on a disk and the maximum pathname length of 64 characters.

The root directory is identical to the pre-2.0 directory. Pre-2.0 disks appear to have only a root directory that contains files but no subdirectories.

Table 1.8 lists the MS-DOS function requests for managing directories.

Table 1.8 Directory-Related Function Requests

39H	Create Directory	Creates a subdirectory.
3AH	Remove Directory	Deletes a subdirectory.
3BH	Change Current Directory	Changes the current directory.
41H	Delete Directory Entry (Unlink)	Deletes a file.
43H	Get/Set File Attributes (Chmod)	Retrieves or changes the attributes of a file.
47H	Get Current Directory	Returns current directory for a given drive.
4EH	Find First File	Searches a directory for the first entry that matches a filename.
4FH	Find Next File	Searches a directory for the next entry that matches a filename.
56H	Change Directory Entry	Renames a file.
57H	Get/Set Date/Time of File	Changes the time and date of last change in a directory entry.

1.5.5 Directory Entry

A directory entry is a 32-byte record that includes the file's name, extension, date and time of last change, and size. An entry in a subdirectory is identical to an entry in the root directory. The directory entry is described in detail in Chapter 3.

1.5.6 File Attributes

Table 1.9 describes the file attributes and how they are represented in the attribute byte of the directory entry (offset 0BH). The attributes can be inspected or changed with Function 43H (Get/Set File Attributes).

Table 1.9 File Attributes

Code	Description
00H	Normal. Can be read or written without restriction.
01H	Read-only. Cannot be opened for write; a file with the same name cannot be created.
02H	Hidden. Not found by directory search.
04H	System. Not found by directory search.
08H	Volume-ID. Only one file can have this attribute; it must be in the root directory.
10H	Subdirectory.
20H	Archive. Set whenever the file is changed, cleared by the Backup command.

The Volume-ID (08H) and Directory (10H) attributes cannot be changed with Function 43H (Get/Set File Attributes).

1.6 MICROSOFT NETWORKS

A Microsoft Network consists of a server and one or more workstations. MS-DOS maintains an assign list that keeps track of which workstation drives and devices have been redirected to the server. For a description of operation and use of the network, see the Microsoft Networks Manager's Guide, and User's Guide.

Table 1.10 lists the MS-DOS function requests for managing a Microsoft Networks workstation.

Table 1.10 Microsoft Network Function Requests

4409H	IOCTL Is Redirected Block	Checks whether a drive letter refers to a local or redirected drive.
440AH	IOCTL Is Redirected Handle	Checks whether a device name refers to a local or redirected device.
5E00H	Get Machine Name	Gets the network name of the workstation.
5E02H	Printer Setup	Defines a string of control characters to be added at the beginning of each file sent to a network printer.
5F02H	Get Assign List Entry	Gets an entry from the assign list that shows the workstation drive letter or device name and the net name of the directory or device on the server to which it is reassigned.
5F03H	Make Assign List Entry	Redirects a workstation drive or device to a server directory or device.
5F04H	Cancel Assign List Entry	Cancels the redirection of a workstation drive or device to a server directory or device.

1.7 MISCELLANEOUS SYSTEM MANAGEMENT

The remaining system calls manage other system functions and resources such as drives, the clock, and addresses. Table 1.11 lists the MS-DOS function requests for managing miscellaneous system resources and operation.

Table 1.11 Miscellaneous System-Management Function Requests

0DH	Reset Disk	Empties all file buffers.
0EH	Select Disk	Sets the default drive.
19H	Get Current Disk	Returns the default drive.
1AH	Set Disk Transfer Address	Establishes the disk I/O buffer.
1BH	Get Default Drive Data	Returns disk format data.
1CH	Get Drive Data	Returns disk format data.
25H	Set Interrupt Vector	Sets interrupt handler address.
29H	Parse File Name	Checks string for valid filename.
2AH	Get Date	Returns system date.
2BH	Set Date	Sets system date.
2CH	Get Time	Returns system time.
2DH	Set Time	Sets system time.
2EH	Set/Reset Verify Flag	Turns disk verify on or off.
2FH	Get Disk Transfer Address	Returns system disk I/O buffer address.
30H	Get MS-DOS Version Number	Returns MS-DOS version number.
33H	Control-C Check	Returns Control-C check status.
35H	Get Interrupt Vector	Returns address of interrupt handler.
36H	Get Disk Free Space	Returns disk space data.
38H	Get/Set Country Data	Sets current country or retrieves country information.
54H	Get Verify State	Returns status of disk verify.

1.8 OLD SYSTEM CALLS

Most of the system calls that have been superseded deal with files. Table 1.12 lists these old calls and the function requests that have superseded them.

Although MS-DOS still includes these old system calls, they should not be used unless it is imperative that a program maintain backward-compatibility with the pre-2.0 versions of MS-DOS.

Table 1.12 Old System Calls and Their Replacements

Old System Call		Has Been Superseded By	
Function Requests		Function Requests	
00H	Terminate Program	4CH	End Process
0FH	Open File	3DH	Open Handle
10H	Close File	3EH	Close Handle
11H	Search for First Entry	4EH	Find First File
12H	Search for Next Entry	4FH	Find Next File
13H	Delete File	41H	Delete Directory Entry
14H	Sequential Read	3FH	Read Handle
15H	Sequential Write	3DH	Open Handle
16H	Create File	3CH	Create Handle
		5AH	Create Temporary File
		5BH	Create New File
17H	Rename File	56H	Change Directory Entry
21H	Random Read	3FH	Read Handle
22H	Random Write	40H	Write Handle
23H	Get File Size	42H	Move File Pointer
24H	Set Relative Record	42H	Move File Pointer
26H	Create New PSP	4B00H	Load and Execute Program
27H	Random Block Read	3FH	Read Handle
28H	Random Block Write	40H	Write Handle
Interrupts		Function Requests	
20H	Program Terminate	4CH	End Process
27H	Terminate But Stay Resident	31H	Keep Process

1.8.1 File Control Block (FCB)

The old file-related function requests require that a program maintain a File Control Block (FCB) for each file; this control block contains such information as the file's name, size, record length, and pointer to current record. MS-DOS does most of this housekeeping for the newer, handle-oriented function requests.

Some descriptions of the old function requests refer to unopened and opened FCBs. An unopened FCB contains only a drive specifier and filename. An opened FCB contains all fields filled by Function 0FH (Open File).

The Program Segment Prefix (PSP) includes room for two FCBs at offsets 5CH and 6CH. See Chapter 4 for a description of the PSP and how these FCBs are used. Table 1.13 describes the fields of the FCB.

Table 1.13 Format of the File Control Block (FCB)

Offset Hex	Dec	Bytes	Name
00H	0	1	Drive number
01H	1	8	Filename
09H	9	3	Extension
0CH	12	2	Current block
0EH	14	2	Record size
10H	16	4	File size
14H	20	2	Date of last write
16H	22	2	Time of last write
18H	24	8	RESERVED
20H	32	1	Current record
21H	33	4	Relative record

Fields of the FCB

Drive Number (offset 00H): Specifies the disk drive; 1 means drive A and 2 means drive B. If the FCB is used to create or open a file, this field can be set to 0 to specify the default drive; the Open File system call sets the field to the number of the default drive.

Filename (offset 01H): Eight characters, left-aligned and padded (if necessary) with blanks. If you specify a reserved device name (such as PRN), do not put a colon at the end.

Extension (offset 09H): Three characters, left-aligned and padded (if necessary) with blanks. This field can be all blanks (no extension).

Current Block (offset 0CH): Points to the block (group of 128 records) that contains the current record. This field and the Current Record field (offset 20H) make up the record pointer. This field is set to 0 by the Open File system call.

Record Size (offset 0EH): The size of a logical record, in bytes. Set to 128 by the Open File system call. If the record size is not 128 bytes, you must set this field after opening the file.

File Size (offset 10H): The size of the file, in bytes. The first word of this 4-byte field is the low-order part of the size.

Date of Last Write (offset 14H): The date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

```
Offset 15H           Offset 14H
|Y|Y|Y|Y|Y|Y|Y|M|    |M|M|M|D|D|D|D|D|
15              9    8    5 4       0
```

Time of Last Write (offset 16H): The time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:

```
Offset 17H           Offset 16H
|H|H|H|H|H|M|M|M|    |M|M|M|S|S|S|S|S|
15       11 10            5 4       0
```

Reserved (offset 18H): These fields are reserved for use by MS-DOS.

Current Record (offset 20H): Points to one of the 128 records in the current block. This field and the Current Block field (offset 0CH) make up the record pointer. This field is not initialized by the Open File system call. You must set it before doing a sequential read or write to the file.

Relative Record (offset 21H): Points to the currently selected record, counting from the beginning of the file (starting with 0). This field is not initialized by the Open File system call. You must set it before doing a random read or write to the file. If the record size is less than 64 bytes, both words of this field are used; if the record size is 64 bytes or more, only the first three bytes are used.

Note

If you use the FCB at offset 5CH of the Program Segment Prefix, the last byte of the Relative Record field is the first byte of the unformatted parameter area that starts at offset 80H. This is the default Disk Transfer Area.

Extended FCB

The Extended File Control Block is used to create or search for directory entries of files with special attributes. It adds the following 7-byte prefix to the FCB:

Name	Size (bytes)	Offset
Flag byte (FFH)	1	-07H
Reserved	5	-06H
Attribute byte	1	-01H

File attributes are described earlier in this chapter in Section 1.5.6, "File Attributes."

1.9 USING THE SYSTEM CALLS

The remainder of this chapter describes how to use the system calls in application programs, lists all the calls in both numeric and alphabetic order, and describes each call in detail.

1.9.1 Issuing An Interrupt

MS-DOS reserves Interrupts 20H through 3FH for its own use. The table of interrupt handler addresses (vector table) is maintained in locations 80H-FCH. Most of the interrupts have been superseded by function requests. Descriptions of three MS-DOS interrupt handlers (Program Terminate, Control-C, and Critical Error) are included in case you must write your own routines to handle these interrupts.

To issue an interrupt, move any required data into the registers and issue the interrupt.

1.9.2 Calling A Function Request

The function requests call MS-DOS routines to manage system resources. Follow this procedure to call a function request:

1. Move any required data into the registers.

2. Move the function number into AH.

3. Move the action code, if required, into AL.

4. Issue Interrupt 21H.

If your program has a standard Program Segment Prefix, an alternative to issuing Interrupt 21H is to execute a long call to location 50H in the PSP.

Whenever possible, it is recommended that the Interrupt 21H method be used.

One other technique supports earlier calling conventions: move any required data into the registers; move the function number into CL; and execute an intrasegment call to location 05H in the current code segment (this location contains a long call to the MS-DOS function dispatcher). This method can only be used with functions 00H through 24H, and always destroys the contents of AX.

1.9.3 Using The Calls From A High-Level Language

The system calls can be executed from any high-level language whose modules can be linked with assembly language modules. In addition to this general technique:

- You can use the DOSXQQ function of Pascal-86 to call a function request directly.

- Use the CALL statement or USER function to execute the required assembly-language code from the BASIC interpreter.

1.9.4 Treatment Of Registers

When MS-DOS takes control after a function request, it switches to an internal stack. Registers not used to return information (except AX) are preserved. The calling program's stack must be large enough to accommodate the interrupt system -- at least 128 bytes in addition to other needs.

1.9.5 Handling Errors

Most of the newer function requests -- those introduced with version 2.0 or later -- set the Carry flag if there is an error, and identify the specific error by returning a number in AX. Table 1.14 lists these error codes and their meanings.

Table 1.14 Error Codes Returned in AX

Code	Meaning
1	Invalid function code
2	File not found
3	Path not found
4	Too many open files (no open handles left)
5	Access denied
6	Invalid handle
7	Memory control blocks destroyed
8	Insufficient memory
9	Invalid memory block address
10	Invalid environment
11	Invalid format
12	Invalid access code
13	Invalid data
14	RESERVED
15	Invalid drive
16	Attempt to remove the current directory
17	Not same device
18	No more files
19	Disk is write-protected
20	Bad disk unit
21	Drive not ready
22	Invalid disk command
23	CRC error
24	Invalid length (disk operation)
25	Seek error
26	Not an MS-DOS disk
27	Sector not found
28	Out of paper
29	Write fault
30	Read fault

31	General failure
32	Sharing violation
33	Lock violation
34	Wrong disk
35	FCB unavailable
36-49	RESERVED
50	Network request not supported
51	Remote computer not listening
52	Duplicate name on network
53	Network name not found
54	Network busy
55	Network device no longer exists
56	Net BIOS command limit exceeded
57	Network adapter hardware error
58	Incorrect response from network
59	Unexpected network error
60	Incompatible remote adapt
61	Print queue full
62	Queue not full
63	Not enough space for print file
64	Network name was deleted
65	Access denied
66	Network device type incorrect
67	Network name not found
68	Network name limit exceeded
69	Net BIOS session limit exceeded
70	Temporarily paused
71	Network request not accepted
72	Print or disk redirection is paused
73-79	RESERVED
80	File exists
81	RESERVED
82	Cannot make
83	Interrupt 24 failure
84	Out of structures
85	Already assigned
86	Invalid password
87	Invalid parameter
88	Net write fault

To handle error conditions, put the following statement immediately after each call similar to XENIX calls:

```
JC <error>
```

where <error> represents the label of an error-handling routine that gets the specific error condition by checking the value in AX and takes appropriate action.

Some of the older system calls return a value in a register that specifies whether the operation was successful. To handle such errors, check the error code and take the appropriate action.

Extended Error Codes

Newer versions of MS-DOS have added more detailed error messages that cannot be used by programs that use the older system calls. To avoid incompatibility, MS-DOS maps these new error codes to the old error code that most closely matches the new one.

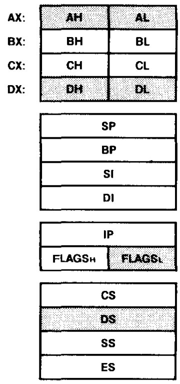
To make use of these new calls, Function 59H (Get Extended Error) has been added. It provides as much detail as possible on the most recent error code returned by MS-DOS. The description of Function 59H lists the new, more detailed error codes and shows how to use this function request.

1.9.6 System Call Descriptions

Most system calls require that information be moved into one or more registers before the call is issued and return information in the registers. The description of each system call in this chapter includes the following:

- A drawing of the 8088 registers that shows their contents before and after the system call.
- A more complete description of the register contents required before the system call.
- A description of the processing performed.
- A more complete description of the register contents after the system call.
- An example of the system call's use.

Figure 1.1 is an example of the drawing of the 8088 registers and how the information is presented.



Call

Return

Figure 1.1 Example of System Call Description

Sample Programs

The sample programs show only data declarations and the code required to use the system calls. Unless stated otherwise, each example assumes a common skeleton that defines the segments and returns control to MS-DOS. Each sample program is intended to be executed as a .COM file. Figure 1.2 shows a complete sample program. The unshaded portion shows what appears in this chapter; the shaded portions are the common skeleton.

```
code        segment
            assume  cs:code,ds:code,es:nothing,ss:nothing
            org     100H
start:      jmp    begin
;
filename    db     "b:\textfile.asc",0
buffer      db      129 dup (?)
handle      dw      ?
;
begin:      open_handle filename,0      ; Open the file
            jc      error_open          ; Routine not shown
            mov     handle,ax           ; Save handle
read_line:  read_handle handle,buffer,128 ; Read 128 bytes
            jc      error_read          ; Routine not shown
            cmp     ax,0                ; End of file?
            je      return              ;   Yes, go home
            mov     bx,ax               ;   No, AX bytes read
            mov     buffer[bx],"$"      ; To terminate string
            display buffer              ; See Function 09H
            jmp     read_line           ; Get next 128 bytes

return:     end_process 0               ; Return to MS-DOS
last_inst:                              ; To mark next byte
;
code        ends
            end     start
```

Figure 1.2 Sample Program With Common Skeleton

To allow the examples to be more complete programs rather than isolated uses of the system calls, a macro is defined for each system call; these macros, plus some general purpose ones, are used in the sample programs. The sample program in the preceding figure includes four such macros: open_handle, read_handle, display, and end_process. All macro definitions are listed at the end of this chapter.

The macros assume the environment for a .COM program as described in Chapter 4; in particular, they assume that all the segment registers contain the same value. To conserve space, the macros generally do not protect registers and leave error checking to the main code. This keeps the macros fairly short, yet useful. You may find such macros a convenient way to include system calls in your assembly language programs.

Error Handling in Sample Programs

Whenever a system call returns an error code, the sample program shows a test for the error condition and a jump to an error routine. To conserve space, the error routines themselves aren't shown. Some error routines might simply display a message and continue processing; in more serious cases, the routine might display a message and end the program (performing any required housekeeping, such as closing files).

Tables 1.15 through 1.18 list the Interrupts and Function Requests in numeric and alphabetic order.

Table 1.15 MS-DOS Interrupts, Numeric Order

Interrupt	Description
20H	Program Terminate
21H	Function Request
22H	Terminate Process Exit Address
23H	Control-C Handler Address
24H	Critical Error Handler Address
25H	Absolute Disk Read
26H	Absolute Disk Write
27H	Terminate But Stay Resident
28H-3FH	RESERVED

Table 1.16 MS-DOS Interrupts, Alphabetic Order

Description	Interrupt
Absolute Disk Read	25H
Absolute Disk Write	26H
Control-C Handler Address	23H
Critical Error Handler Address	24H
Function Request	21H
Program Terminate	20H
RESERVED	28H-3FH
Terminate Process Exit Address	22H
Terminate But Stay Resident	27H

Table 1.17 MS-DOS Function Requests, Numeric Order

Function	Description
00H	Terminate Program
01H	Read Keyboard And Echo
02H	Display Character
03H	Auxiliary Input
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
07H	Direct Console Input
08H	Read Keyboard
09H	Display String
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
0DH	Reset Disk
0EH	Select Disk
0FH	Open File
10H	Close File
11H	Search For First Entry
12H	Search For Next Entry
13H	Delete File
14H	Sequential Read
15H	Sequential Write
16H	Create File
17H	Rename File
18H	RESERVED
19H	Get Current Disk
1AH	Set Disk Transfer Address
1BH	Get Default Drive Data
1CH	Get Drive Data
1DH-20H	RESERVED
21H	Random Read
22H	Random Write
23H	Get File Size
24H	Set Relative Record
25H	Set Interrupt Vector
26H	Create New PSP
27H	Random Block Read
28H	Random Block Write
29H	Parse File Name
2AH	Get Date
2BH	Set Date
2CH	Get Time
2DH	Set Time
2EH	Set/Reset Verify Flag
2FH	Get Disk Transfer Address
30H	Get MS-DOS Version Number
31H	Keep Process
32H	RESERVED
33H	Control-C Check
34H	RESERVED

35H	Get Interrupt Vector
36H	Get Disk Free Space
37H	RESERVED
38H	Get/Set Country Data
39H	Create Directory
3AH	Remove Directory
3BH	Change Current Directory
3CH	Create Handle
3DH	Open Handle
3EH	Close Handle
3FH	Read Handle
40H	Write Handle
41H	Delete Directory Entry
42H	Move File Pointer
43H	Get/Set File Attributes
4400H,4401H	IOCTL Data
4402H,4403H	IOCTL Character
4404H,4405H	IOCTL Block
4406H,4407H	IOCTL Status
4408H	IOCTL Is Changeable
4409H	IOCTL Is Redirected Block
440AH	IOCTL Is Redirected Handle
440BH	IOCTL Retry
45H	Duplicate File Handle
46H	Force Duplicate File Handle
47H	Get Current Directory
48H	Allocate Memory
49H	Free Allocated Memory
4AH	Set Block
4B00H	Load and Execute Program
4B03H	Load Overlay
4CH	End Process
4DH	Get Return Code Child Process
4EH	Find First File
4FH	Find Next File
50H-53H	RESERVED
54H	Get Verify State
55H	RESERVED
56H	Change Directory Entry
57H	Get/Set Date/Time of File
58H	Get/Set Allocation Strategy
59H	Get Extended Error
5AH	Create Temporary File
5BH	Create New File
5C00H	Lock
5C01H	Unlock
5DH	RESERVED
5E00H	Get Machine Name
5E02H	Printer Setup
5F02H	Get Assign List Entry
5F03H	Make Assign List Entry
5F04H	Cancel Assign List Entry
60H-61H	RESERVED
62H	Get PSP
63H-7FH	RESERVED

Table 1.18 MS-DOS Function Requests, Alphabetic Order

Function	Description
48H	Allocate Memory
03H	Auxiliary Input
04H	Auxiliary Output
0AH	Buffered Keyboard Input
5F04H	Cancel Assign List Entry
3BH	Change Current Directory
56H	Change Directory Entry
0BH	Check Keyboard Status
10H	Close File
3EH	Close Handle
33H	Control-C Check
39H	Create Directory
16H	Create File
3CH	Create Handle
5BH	Create New File
26H	Create New PSP
5AH	Create Temporary File
41H	Delete Directory Entry
13H	Delete File
06H	Direct Console I/O
07H	Direct Console Input
02H	Display Character
09H	Display String
45H	Duplicate File Handle
4CH	End Process
4EH	Find First File
4FH	Find Next File
0CH	Flush Buffer, Read Keyboard
46H	Force Duplicate File Handle
49H	Free Allocated Memory
5F02H	Get Assign List Entry
47H	Get Current Directory
19H	Get Current Disk
2AH	Get Date
1BH	Get Default Drive Data
36H	Get Disk Free Space
2FH	Get Disk Transfer Address
1CH	Get Drive Data
59H	Get Extended Error
23H	Get File Size
35H	Get Interrupt Vector
5E01H	Get Machine Name
30H	Get MS-DOS Version Number
62H	Get PSP
4DH	Get Return Code Of Child Process
2CH	Get Time
54H	Get Verify State
58H	Get/Set Allocation Strategy
38H	Get/Set Country Data
57H	Get/Set Date/Time Of File

43H	Get/Set File Attributes
4404H,4405H	IOCTL Block
4402H,4403H	IOCTL Character
4400H,4401H	IOCTL Data
4408H	IOCTL Is Changeable
4409H	IOCTL Is Redirected Block
440AH	IOCTL Is Redirected Handle
440BH	IOCTL Retry
4406H,4407H	IOCTL Status
31H	Keep Process
4B00H	Load and Execute Program
4B03H	Load Overlay
5C00H	Lock
5F03H	Make Assign List Entry
42H	Move File Pointer
0FH	Open File
3DH	Open Handle
29H	Parse File Name
05H	Print Character
5E02H	Printer Setup
27H	Random Block Read
28H	Random Block Write
21H	Random Read
22H	Random Write
3FH	Read Handle
08H	Read Keyboard
01H	Read Keyboard And Echo
3AH	Remove Directory
17H	Rename File
18H	RESERVED
1BH-20H	RESERVED
32H	RESERVED
34H	RESERVED
37H	RESERVED
50H-53H	RESERVED
55H	RESERVED
60H-61H	RESERVED
63H-7FH	RESERVED
0DH	Reset Disk
11H	Search For First Entry
12H	Search For Next Entry
0EH	Select Disk
14H	Sequential Read
15H	Sequential Write
4AH	Set Block
2BH	Set Date
1AH	Set Disk Transfer Address
25H	Set Interrupt Vector
24H	Set Relative Record
2DH	Set Time
2EH	Set/Reset Verify Flag
00H	Terminate Program
5C01H	Unlock
40H	Write Handle

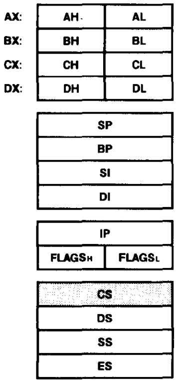
A detailed description of each system call follows. They are listed in numeric order; the interrupts are described first, then the function requests.

Note: Unless otherwise stated, all numbers in the system call descriptions--both text and code--are in hexadecimal.

1.10 INTERRUPTS

The following pages describe Interrupts 20H-27H.

Program Terminate (Interrupt 20H)



Call
CS
Segment address of Program Segment Prefix

Return
None

Interrupt 20H terminates the current process and returns control to its parent process. All open file handles are closed and the disk cache is cleaned. CS must contain the segment address of the Program Segment Prefix when this interrupt is issued.

Interrupt 20H is provided only for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function Request 4CH, End Process, which permits returning a completion code to the parent process and does not require CS to contain the segment address of the Program Segment Prefix.

The following exit addresses are restored from the Program Segment Prefix:

Offset	Exit Address
0AH	Program terminate
0EH	Control-C
12H	Critical error

Note

Close all files that have changed in length before issuing this interrupt. If a changed file is not closed, its length is not recorded correctly in the directory. See Functions 10H and 3EH for a description of the Close File system calls.

Macro Definition:

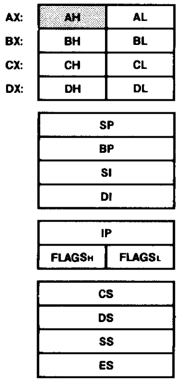
```
terminate macro
          int 20H
          endm
```

Example

The following program displays a message and returns to MS-DOS. It uses only the opening portion of the sample program skeleton shown in Figure 1.2:

```
message db "displayed by INT20H example". 0DH, 0AH, "$"
;
begin:  display message   ;see Function 09H
        terminate         ;THIS INTERRUPT
code    ends
        end     start
```

Function Request (Interrupt 21H)



Call
AH
  Function number

Other registers
  As specified in individual function

Return
As specified in individual function

Interrupt 21H causes MS-DOS to carry out the function request whose number is in AH. See Section 1.11, "Function Requests," for a description of the MS-DOS functions.

Example

To call the Get Time function:

```
     mov    ah,2CH              ;Get Time is Function 2CH
     int    21H                 ;MS-DOS function request
```

Terminate Process Exit Address (Interrupt 22H)

When a program terminates, MS-DOS transfers control to the routine that starts at the address in the Interrupt 22H entry in the vector table. When MS-DOS creates a program segment, it copies this address into the PSP starting at offset 0AH.

This interrupt must never be issued by a user program; it is issued only by MS-DOS. If you must write your own terminate interrupt handler, use Function Request 35H (Get Interrupt Vector) to get the address of the standard routine, save the address, then use Function Request 25H (Set Interrupt Vector) to change the Interrupt 22H entry in the vector table to point to your routine.

Control-C Handler Address (Interrupt 23H)

When a user types Control-C or Control-Break (on IBM-compatibles), MS-DOS transfers control as soon as possible to the routine that starts at the address in the Interrupt 23H entry in the vector table. When MS-DOS creates a program segment, it copies the address currently in the interrupt table into the PSP starting at offset 0EH.

This interrupt must never be issued by a user program; it is issued only by MS-DOS. If you must write your own Control-C interrupt handler, use Function Request 35H (Get Interrupt Vector) to get the address of the standard routine, save the address, then use Function Request 25H (Set Interrupt Vector) to change the Interrupt 23H entry in the vector table to point to your routine.

If the Control-C routine preserves all registers, it can end with an IRET instruction (return from interrupt) to continue program execution. If the user-written interrupt program returns with a long return, the carry flag is used to determine whether or not the program will abort. If the carry flag is set, it will be aborted; otherwise, execution will continue as with a return by IRET. If the user-written Control-Break interrupt uses function calls 09H or 0AH, then Control-C, Return, and Line Feed are output. If execution continues with an IRET instruction, I/O continues from the start of the line.

When the interrupt occurs, all registers are set to the value they had when the original call to MS-DOS was made. There are no restrictions on what a Control-C handler can do -- including MS-DOS function calls -- as long as the registers are unchanged if IRET is used.

If Function 09H or 0AH (Display String or Buffered Keyboard Input) is interrupted by Control-C, the three-byte sequence 03H-0DH-0AH (usually displayed as C followed by a carriage return) is sent to the display and the function resumes at the beginning of the next line.

If a program creates a second PSP and executes a second program -- using Function 4B00H (Load and Execute Program), for example -- and the second program changes the Control-C address in the vector table, MS-DOS restores the Control-C vector to its original value before returning control to the calling program.

Critical Error Handler Address (Interrupt 24H)

If a critical error occurs during execution of an I/O function request -- this usually means a fatal disk error -- MS-DOS transfers control to the routine that starts at the address in the Interrupt 24H entry in the vector table. When MS-DOS creates a program segment, it copies this address into the PSP starting at offset 12H.

This interrupt must never be issued by a user program; it is issued only by MS-DOS. If you must write your own critical error interrupt handler, use Function Request 35H (Get Interrupt Vector) to get the address of the standard routine, save the address, then use Function Request 25H (Set Interrupt Vector) to change the Interrupt 24H entry in the vector table to point to your routine.

Interrupt 24H is not issued if a failure occurs during execution of Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write). These errors are handled by the error routine in COMMAND.COM that retries the disk operation, then gives the user the choice of aborting, retrying the operation, or ignoring the error.

The following topics describe the requirements of an Interrupt 24H routine, the error codes, registers, and stack.

1.10.1 Conditions Upon Entry

After retrying an I/O error three times, MS-DOS issues Interrupt 24H. The interrupt handler receives control with interrupts disabled. AX and DI contain error codes, and BP contains the offset (to the segment address in SI) of a Device Header control block that describes the device on which the error occurred.

1.10.2 Requirements For An Interrupt 24H Handler

To use the MS-DOS critical error handler to issue the "Abort, Retry, or Ignore" prompt and get the user's response, the first thing a user-written critical error handler should do is push the flags and execute a far call to the address of the standard Interrupt 24H handler (the user program that changed the Interrupt 24H vector should have saved this address). After the user responds to the prompt, MS-DOS returns control to the user-written routine.

NOTE: There are source applications which will have trouble with this as it changes the stack frame.

The error handler can do its processing now, but before it does anything else it must preserve BX, CX, DX, DS, ES, SS, and SP. Only function calls 01-0CH inclusive and 59H may be used (if it uses any others, the MS-DOS stack is destroyed and MS-DOS is left in an unpredictable state), nor should it change the contents of the Device Header.

If an Interrupt 24H routine returns to the user program (rather than returning to MS-DOS), it must restore the user program's registers -- removing all but the last three words from the stack -- and issue an IRET. Control returns to the statement immediately following the I/O function request that resulted in the error. This leaves MS-DOS in an unstable state until a function request above 0CH is called.

User Stack

The user stack is in effect, and contains the following (starting with the top of the stack):

```
IP      MS-DOS registers from issuing Interrupt 24H
CS
FLAGS

AX      User registers at time of original
BX      INT 21H
CX
DX
SI
DI
BP
DS
ES

IP      From the original INT 21H
CS      from the user to MS-DOS
FLAGS
```

The registers are set such that if the user-written error handler issues an IRET, MS-DOS responds according to the value in AL:

AL	Action
0	Ignore the error.
1	Retry the operation.
2	Abort the program by issuing Interrupt 23H.
3	Fail the system call that is in progress.

Note that the ignore option may cause unexpected results as it causes MS-DOS to believe that an operation completed successfully when it didn't.

Disk Error Code in AX

If bit 7 of AH is 0, the error occurred on a disk drive. AL contains the failing drive (0=A, 1=B, etc.). Bit 0 of AH specifies whether the error occurred during a read or write operation (0=read, 1=write), and bits 1 and 2 of AH identify the area of the disk where the error occurred:

Bits 2-1	Location of error
00	MS-DOS area
01	File Allocation Table
10	Directory
11	Data area

Bits 3-5 of AH specify valid responses to the error prompt:

Bit	Value	Response
3	0	Fail not allowed
	1	Fail allowed
4	0	Retry not allowed
	1	Retry allowed
5	0	Ignore not allowed
	1	Ignore allowed

If Retry is specified but not allowed, MS-DOS changes it to Fail. If Ignore is specified but not allowed, MS-DOS changes it to Fail. If Fail is specified but not allowed, MS-DOS changes it to Abort. The Abort response is always allowed.

Other Device Error Code in AX

If bit 7 of AH is 1, either the memory image of the File Allocation Table (FAT) is bad or an error occurred on a character device. The device header pointed to by BP:SI contains a word of attribute bits that identify the type of device and, therefore, the type of error.

The word of attribute bits is at offset 04H of the Device Header. Bit 15 specifies the type of device (0=block, 1=character).

If bit 15 is 0 (block device), the error was a bad memory image of the FAT.

If bit 15 is 1 (character device), the error was on a character device. DI contains the error code, the contents of AL are undefined, and bits 0-3 of the attribute word have the following meaning:

Bit	Meaning If Set
0	Current standard input
1	Current standard output
2	Current null device
3	Current clock device

See Chapter 2 for a complete description of the Device Header control block.

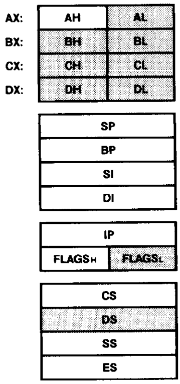
Error Code in DI

The high byte of DI is undefined. The low byte contains the following error codes:

Error Code	Description
0	Attempt to write on write-protected disk
1	Unknown unit
2	Drive not ready
3	Unknown command
4	CRC error in data
5	Bad drive request structure length
6	Seek error
7	Unknown media type
8	Sector not found
9	Printer out of paper
A	Write fault
B	Read fault
C	General failure

A user-written Interrupt 24H handler can use Function 59H (Get Extended Error) to get detailed information about the error that caused the interrupt to be issued.

Absolute Disk Read (Interrupt 25H)



Call
AL
  Drive number
DS:BX
  Disk Transfer Address
CX
  Number of sectors
DX
  Beginning relative sector

Return
AL
  Error code if CF=1
FlagsL
  CF = 0 if successful
     = 1 if not successful

The registers must contain the following:

AL	Drive number (0=A, 1=B, etc.).
BX	Offset of Disk Transfer Address (from segment address in DS).
CX	Number of sectors to read.
DX	Beginning relative sector.

Warning

It is strongly recommended that the use of this function be avoided unless absolutely necessary. Access to files should be done through the normal MS-DOS function requests. There is no guarantee of upward compatibility for the Absolute Disk I/O in future releases of MS-DOS.

This interrupt transfers control to the device driver. The number of sectors specified in CX is read from the disk to the Disk Transfer Address. Its requirements and processing are identical to Interrupt 26H, except data is read rather than written. Very little checking is done on the user's input parameters; therefore, care must be used to make sure they are reasonable. Failure to do this may cause strange results or a system crash.

Note

All registers except the segment registers are destroyed by this call. Be sure to save any registers your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. Be sure to pop the stack upon return to prevent uncontrolled growth.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H earlier in this section for the codes and their meanings).

Macro Definition:

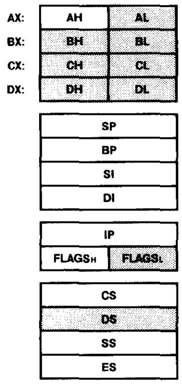
```
abs_disk_read  macro  disk,buffer,num_sectors,first_sector
               mov    al,disk
               mov    bx,offset buffer
               mov    cx,num_sectors
               mov    dx,first_sector
               int    25H
               popf
               endm
```

Example

The following program copies the contents of a single-sided disk in drive A to the disk in drive B.

```
prompt     db    "Source in A, target in B",0DH,0AH
           db    "Any key to start. $"
first      dw    0
buffer     db    60 dup (512 dup (?))   ;60 sectors
;
begin:     display prompt          ;see Function 09H
           read_kbd                ;see Function 08H
           mov      cx,6           ;copy 6 groups of
                                     ;60 sectors
copy:      push     cx             ;save the loop counter
           abs_disk_read  0,buffer,60,first   ;THIS INTERRUPT
           abs_disk_write 1,buffer,60,first   ;see INT 26H
           add  first,60           ;do the next 60 sectors
           pop  cx                 ;restore the loop counter
           loop copy
```

Absolute Disk Write (Interrupt 26H)



Call
AL
  Drive number
DS:BX
  Disk Transfer Address
CX
  Number of sectors
DX
  Beginning relative sector

Return
AL
  Error code if CF = 1
FLAGSL
  CF = 0 if successful
       1 if not successful

> **Warning**
>
> It is strongly recommended that the use of this function be avoided unless absolutely necessary. Access to files should be done through the normal MS-DOS function requests. There is no guarantee of upward compatibility for the Absolute Disk I/O in future releases of MS-DOS.

The registers must contain the following:

AL	Drive number (0=A, 1=B, etc.).
BX	Offset of Disk Transfer Address (from segment address in DS).
CX	Number of sectors to write.
DX	Beginning relative sector.

This interrupt transfers control to MS-DOS. The number of sectors specified in CX is written from the Disk Transfer Address to the disk. Its requirements and processing are identical to Interrupt 25H, except data is written to the disk rather than read from it. Very little checking is done on the user's input parameters; therefore, care must be used to make sure they are reasonable. Failure to do this may cause strange results or a system crash.

Note

All registers except the segment registers are destroyed by this call. Be sure to save any registers your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. Be sure to pop the stack upon return to prevent uncontrolled growth.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H for the codes and their meanings).

Macro Definition:

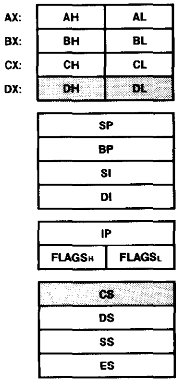
```
abs_disk_write  macro  disk,buffer,num_sectors,first_sector
                mov    al,disk
                mov    bx,offset buffer
                mov    cx,num_sectors
                mov    dx,first_sector
                int    26H
                popf
                endm
```

Example

The following program copies the contents of a single-sided disk in drive A to the disk in drive B, verifying each write. It uses a buffer of 32K bytes.

```
off         equu   0
on          equ    1
;
prompt      db     "Source in A, target in B",0DH,0AH
            db     "Any key to start. $"
first       dw     0
buffer      db     60 dup (512 dup (?))  ;60 sectors
;
begin:      display prompt         ;see Function 09H
            read_kbd               ;see Function 08H
            verify  on             ;see Function 2EH
            mov     cx,6           ;copy 6 groups of 60 sectors
copy:       push    cx             ;save the loop counter
            abs_disk_read  0,buffer,60,first  ;see INT 25H
            abs_disk_write 1,buffer,60,first  ;THIS INTERRUPT
            add  first,60          ;do the next 60 sectors
            pop  cx                ;restore the loop counter
            loop copy
            verify  off            ;see Function 2EH
```

Terminate But Stay Resident (Interrupt 27H)



Call
CS:DX
  Pointer to first byte following
  last byte of code.

Return
None

Interrupt 27H makes a program up to 64K in size remain resident after it terminates. It is often used to install device-specific interrupt handlers.

This interrupt is provided only for compatibility with versions of MS-DOS prior to 2.0. You should use Function 31H (Keep Process), which lets programs larger than 64K remain resident and allows return information to be passed, to install a resident program unless it is absolutely imperative that your program be compatible with pre-2.0 versions of MS-DOS.

DX must contain the offset (from the segment address in CS) of the first byte following the last byte of code in the program. When Interrupt 27H is executed, the program terminates and control returns to DOS, but the program is not overlaid by other programs. Files left open are not closed. When the interrupt is called, CS must contain the segment address of the Program Segment Prefix (the value of DS and ES when execution started).

This interrupt must not be used by .EXE programs that are loaded into high memory. It restores the Interrupt 22H, 23H, and 24H vectors, so it cannot be used to install new Control-C or critical error handlers.

Macro Definition:

```
stay_resident macro last_instruc
              mov   dx,offset last_instruc
              inc   dx
              int   27H
              endm
```

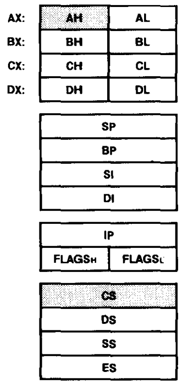
Example

Because the most common use of this call is to install a machine-specific routine, an example is not shown. The macro definition shows the calling syntax.

1.11 FUNCTION REQUESTS

The following pages describe function calls 00H-62H.

Terminate Program (Function 00H)



Call
AH = 00H
CS
  Segment address of
  Program Segment Prefix

Return
None

Function 00H is called by Interrupt 20H; it performs the same processing.

The CS register must contain the segment address of the Program Segment Prefix before you call this interrupt.

The following exit addresses are restored from the specified offsets in the Program Segment Prefix:

Offset	Exit Address
0AH	Program terminate
0EH	Control-C
12H	Critical error

All file buffers are flushed to disk.

Warning

Close all files that have changed in length before calling this function. If a changed file is not closed, its length is not recorded correctly in the directory. See Function 10H for a description of the Close File system call.

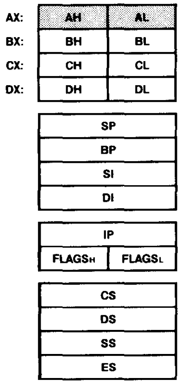
```
Macro Definition:  terminate_program  macro
                                      xor     ah,ah
                                      int     21H
                                      endm
```

Example

The following program displays a message and returns to MS-DOS. It uses only the opening portion of the sample program skeleton shown in Figure 1.2.

```
message db "Displayed by FUNC00H example", 0DH,0AH,"$"
;
begin:  display message     ;see Function 09H
        terminate_program   ;THIS FUNCTION
code    ends
        end     start
```

Read Keyboard and Echo (Function 01H)



Call
AH = 01H

Return
AL
  Character typed

Function 01H waits for a character to be read from standard input, then echoes the character to standard output and returns it in AL. If the character is Control-C, Interrupt 23H is executed.

Macro Definition:

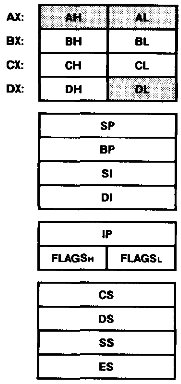
```
read_kbd_and_echo  macro
                   mov  ah, 01H
                   int  21H
                   endm
```

Example

The following program displays and prints characters as they are typed. If Return is pressed, the program sends a Line Feed-Carriage Return sequence to both the display and the printer.

```
begin:     read_kbd_and_echo          ;THIS FUNCTION
           print_char    al           ;see Function 05H
           cmp           al,0DH       ;is it a CR?
           jne           begin        ;no, print it
           print_char    0AH          ;see Function 05H
           display_char  0AH          ;see Function 02H
           jmp           begin        ;get another character
```

Display Character (Function 02H)



Call
AH = 02H
DL
  Character to be displayed

Return
None

Function 02H sends the character in DL to standard output. If Control-C is typed, Interrupt 23H is issued.

Macro Definition:

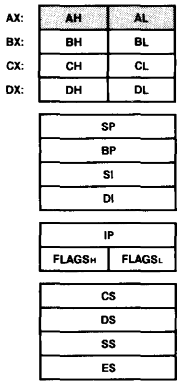
```
display_char macro   character
             mov     dl,character
             mov     ah,02H
             int     21H
             endm
```

Example

The following program converts lowercase characters to uppercase before displaying them.

```
begin:      read_kbd                    ;see Function 08H
            cmp     al,"a"
            jl      uppercase           ;don't convert
            cmp     al,"z"
            jg      uppercase           ;don't convert
            sub     al,20H              ;convert to ASCII code
                                        ;for uppercase
uppercase: display_char al              ;THIS FUNCTION
            jmp     begin:              ;get another character
```

Auxiliary Input (Function 03H)



Call
AH = 03H

Return
AL
  Character from auxiliary device

Function 03H waits for a character from standard auxiliary, then returns the character in AL. This system call does not return a status or error code.

If a Control-C has been typed at console input, Interrupt 23H is issued.

Macro Definition:

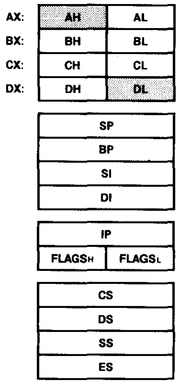
```
aux_input  macro
           mov   ah,03H
           int   21H
           endm
```

Example

The following program prints characters as they are received from the auxiliary device. It stops printing when an end-of-file character (ASCII 26, or Control-Z) is received.

```
begin:    aux_input            ;THIS FUNCTION
          cmp   al,1AH         ;end of file?
          je    return         ;yes, all done
          print_char  al       ;see Function 05H
          jmp   begin          ;get another character
```

Auxiliary Output (Function 04H)



Call
AH = 04H
DL
  Character for auxiliary device

Return
None

Function 04H sends the character in DL to standard auxiliary. This system call does not return a status or error code.

If a Control-C has been typed at console input, Interrupt 23H is issued.

Macro Definition:

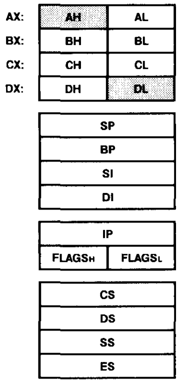
```
aux_output  macro  character
            mov   dl,character
            mov   ah,04H
            int   21H
            endm
```

Example

The following program gets a series of strings of up to 80 bytes from the keyboard, sending each to the auxiliary device. It stops when a null string (CR only) is typed.

```
string    db    81 dup(?) ;see Function 0AH
;
begin:    get_string  80,string         ;see Function 0AH
          cmp  string[1],0              ;null string?
          je   return                   ;yes, all done
          mov  cx, word ptr string[1]   ;get string length
          mov  bx,0                     ;set index to 0
send_it:  aux_output string[bx+2]       ;THIS FUNCTION
          inc  bx                       ;bump index
          loop  send_it                 ;send another character
          jmp  begin                    ;get another string
```

Print Character (Function 05H)



Call
AH = 05H
DL
  Character for printer

Return
None

Function 05H sends the character in DL to the standard printer. If Control-C has been typed at console input, Interrupt 23H is issued. This function request does not return a status or error code.

Macro Definition:

```
print_char macro character
           mov   dl,character
           mov   ah,05H
           int   21H
           endm
```

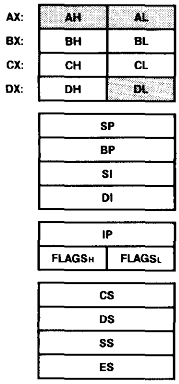
Example

The following program prints a walking test pattern on the printer. It stops if Control-C is pressed.

```
line_num     db     0
;
begin:       mov    cx,60          ;print 60 lines
start_line:  mov    bl,33          ;first printable ASCII
                                   ;character (!)
             add    bl,line_num    ;to offset one character
             push   cx             ;save number-of-lines counter
             mov    cx,80          ;loop counter for line
print_it:    print_char bl         ;THIS FUNCTION
             inc    bl             ;move to next ASCII character
             cmp    bl,126         ;last printable ASCII
                                   ;character (~)
             jl     no_reset       ;not there yet
```

```
no_reset:   loop  print_it        ;print another character
            print_char 0DH        ;carriage return
            print_char 0AH        ;line feed
            inc   line_num        ;to offset 1st char. of line
            pop   cx              ;restore #-of-lines counter
            loop  start_line      ;print another line
```

Direct Console I/O (Function 06H)



Call
AH = 06H
DL
  See text

Return
AL
  If DL = FFH before call,
  then zero flag not set means AL
  has character from standard input.
  Zero flag set means there was not
  a character to get, and AL = 0

The action of Function 06H depends on the value in DL when the function is called:

Value in DL	Action
FFH	If a character has been read from standard input, it is returned in AL and the zero flag is cleared (0); if a character has not been read, the zero flag is set (1).
Not FFH	The character in DL is sent to standard output.

This function does not check for Control-C.

Macro Definition:

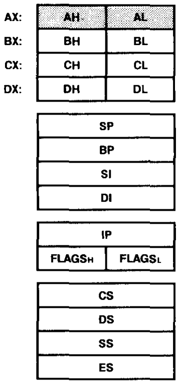
```
dir_console_io  macro switch
                mov   dl,switch
                mov   ah,06H
                int   21H
                endm
```

Example

The following program sets the system clock to 0 and continuously displays the time. When any character is typed, the display freezes; when any character is typed again, the clock is reset to 0 and the display starts again.

```
time            db  "00:00:00.00",0DH,0AH,"$"  ;see Function 09H
;                                             ;for explanation of $
;
begin:          set_time  0,0,0,0           ;see Function 2DH
read_clock:     get_time                    ;see Function 2CH
                byte_to_dec  ch,time        ;see end of chapter
                byte_to_dec  cl,time[3]     ;see end of chapter
                byte_to_dec  dh,time[6]     ;see end of chapter
                byte_to_dec  dl,time[9]     ;see end of chapter
                display  time               ;see Function 09H
                dir_console_io  FFH         ;THIS FUNCTION
                cmp       al,0              ;character typed?
                jne       stop              ;yes, stop timer
                jmp       read_clock        ;no, keep timer
                                            ;running
stop:           read_kbd                    ;see Function 08H
                jmp       begin             ;start over
```

Direct Console Input (Function 07H)



Call
AH = 07H

Return
AL
Character from keyboard

Function 07H waits for a character to be read from standard input, then returns it in AL. This function does not echo the character or check for Control-C. (For a keyboard input function that echoes or checks for Control-C, see Function 01H or 08H.)

Macro Definition:

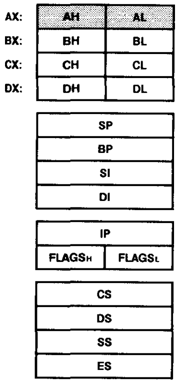
```
dir_console_input  macro
                   mov  ah,07H
                   int  21H
                   endm
```

Example

The following program prompts for a password (8 characters maximum) and places the characters into a string without echoing them.

```
password  db  8 dup(?)
prompt    db "Password: $"         ;see Function 09H for
                                    ;explanation of $
begin:    display prompt            ;see Function 09H
          mov  cx,8                 ;maximum length of password
          xor  bx,bx                ;so BL can be used as index
get_pass: dir_console_input         ;THIS FUNCTION
          cmp  al,0DH               ;was it a CR?
          je   return               ;yes, all done
          mov  password[bx],al      ;no, put character in string
          inc  bx                   ;bump index
          loop get_pass             ;get another character
```

Read Keyboard (Function 08H)



Call
AH = 08H

Return
AL
  Character from keyboard

Function 08H waits for a character to be read from standard input, then returns it in AL. If Control-C is pressed, Interrupt 23H is executed. This function does not echo the character. (For a keyboard input function that echoes the character or checks for Control-C, see Function 01H.)

Macro Definition:

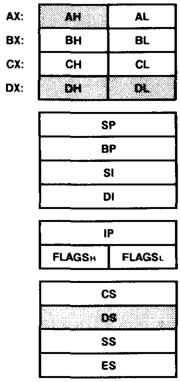
```
read_kbd  macro
          mov   ah,08H
          int   21H
          endm
```

Example

The following program prompts for a password (8 characters maximum) and places the characters into a string without echoing them.

```
password  db  8 dup(?)
prompt    db "Password: $"        ;see Function 09H
                                  ;for explanation of $
begin:    display prompt          ;see Function 09H
          mov  cx,8               ;maximum length of password
          xor  bx,bx              ;BL can be an index
get_pass: read_kbd                ;THIS FUNCTION
          cmp  al,0DH             ;was it a CR?
          je   return             ;yes, all done
          mov  password[bx],al    ;no, put char. in string
          inc  bx                 ;bump index
          loop get_pass           ;get another character
```

Display String (Function 09H)



Call
AH = 09H
DS:DX
  Pointer to string to be displayed

Return
None

Function 09H sends to standard output a string that ends with "$" (the $ is not displayed). DX must contain the offset (from the segment address in DS) of the string.

Macro Definition:

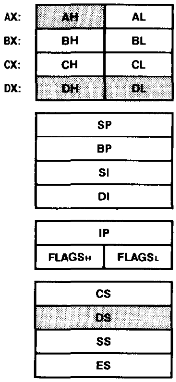
```
display  macro string
         mov   dx,offset string
         mov   ah,09H
         int   21H
         endm
```

Example

The following program displays the hexadecimal code of the key that is typed.

```
table    db      "0123456789ABCDEF"
result   db      " - 00H",0DH,0AH,"$"  ;see text for
                                       ;explanation of $
begin:   read_kbd_and_echo             ;see Function 01H
         xor     ah,ah                 ;clear upper byte
         convert ax,16,result[3] ;see end of chapter
         display result                ;THIS FUNCTION
         jmp     begin                 ;do it again
```

Buffered Keyboard Input (Function 0AH)



Call
AH = 0AH
DS:DX
  Pointer to input buffer

Return
None

Function 0AH gets a string from standard input. DX must contain the offset (from the segment address in DS) of an input buffer of the following form:

Byte	Contents
1	Maximum number of characters in buffer, including the carriage return (you must set this value).
2	Actual number of characters typed, not counting the Carriage Return (the function sets this value).
3-n	Buffer; must be at least as long as the number in byte 1.

Characters are read from standard input and placed in the buffer beginning at the third byte until a Return (0DH) is read. If the buffer fills to one less than the maximum, additional characters read are ignored and 07H (Bel) is sent to standard output until a Return is read. If the string is typed at the console, it can be edited as it is being entered. If Control-C is typed, Interrupt 23H is issued.

MS-DOS sets the second byte of the buffer to the number of characters read (not counting the Carriage Return).

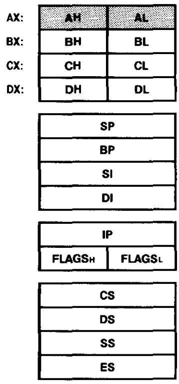
```
Macro Definition:  get_string  macro  limit,string
                               mov    dx,offset string
                               mov    string,limit
                               mov    ah,0AH
                               int    21H
                               endm
```

Example

The following program gets a 16-byte (maximum) string from the keyboard and fills a 24-line by 80-character screen with it.

```
buffer            label byte
max_length        db     ?                 ;maximum length
chars_entered     db     ?                 ;number of chars.
string            db     17 dup (?)        ;16 chars + CR
strings_per_line  dw     0                 ;how many strings
                                           ;fit on line
crlf              db     0DH,0AH
;
begin:            get_string 17,buffer     ;THIS FUNCTION
                  xor   bx,bx              ;so byte can be
                                           ;used as index
                  mov   bl,chars_entered   ;get string length
                  mov   buffer[bx+2],"$"   ;see Function 09H
                  mov   al,50H             ;columns per line
                  cbw
                  div   chars_entered      ;times string fits
                                           ;on line
                  xor   ah,ah              ;clear remainder
                  mov   strings_per_line,ax ;save col. counter
                  mov   cx,24              ;row counter
display_screen:   push cx                  ;save it
                  mov   cx,strings_per_line ;get col. counter
display_line:     display  string          ;see Function 09H
                  loop display_line
                  display crlf             ;see Function 09H
                  pop   cx                 ;get line counter
                  loop display_screen      ;display 1 more line
```

Check Keyboard Status (Function 0BH)



Call
AH = 0BH

Return
AL
FFH = characters in type-ahead buffer
0 = no characters in type-ahead buffer

Function 0BH checks whether characters are available from standard input (if standard input has not been redirected, the type-ahead buffer). If characters are available, AL returns FFH; if not, AL returns 0. If Control-C is in the buffer, Interrupt 23H is executed.

Macro Definition:

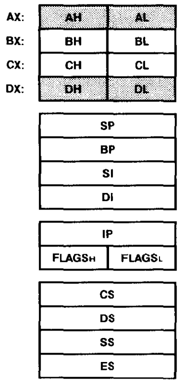
```
check_kbd_status  macro
                  mov   ah,0BH
                  int   21H
                  endm
```

Example

The following program continuously displays the time until any key is pressed.

```
time        db       "00:00:00.00",0DH,0AH,"$"
            .
            .
begin:      get_time                     ;see Function 2CH
            byte_to_dec ch,time          ;see end of chapter
            byte_to_dec cl,time[3]       ;see end of chapter
            byte_to_dec dh,time[6]       ;see end of chapter
            byte_to_dec dl,time[9]       ;see end of chapter
            display time                 ;see Function 09H
            check_kbd_status             ;THIS FUNCTION
            cmp      al,0FFH             ;has a key been typed?
            je       return              ;yes, go home
            jmp      begin               ;no, keep displaying
                                         ;time
```

Flush Buffer, Read Keyboard (Function 0CH)



Call
AH = 0CH
AL
  1, 6, 7, 8, or 0AH = the corresponding function is called.
  Any other value = no further processing.

Return
AL
  0 = Type-ahead buffer was flushed; no other processing performed.

Function 0CH empties the standard input buffer (if standard input has not been redirected, Function 0CH empties the type-ahead buffer). Further processing depends on the value in AL when the function is called.

1, 6, 7, 8, or 0AH -- The corresponding MS-DOS function is executed.

Any other value -- No further processing; AL returns 0.

Macro Definition:

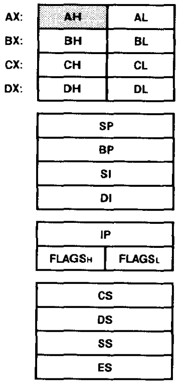
```
flush_and_read_kbd  macro switch
                    mov   al,switch
                    mov   ah,0CH
                    int   21H
                    endm
```

Example

The following program both displays and prints characters as they are typed. If Return is pressed, the program sends Carriage Return-Line Feed to both the display and the printer.

```
begin:      flush_and_read_kbd 1        ;THIS FUNCTION
            print_char    al            ;see Function 05H
            cmp           al,0DH        ;is it a CR?
            jne           begin         ;no, print it
            print_char    0AH           ;see Function 05H
            display_char  0AH           ;see Function 02H
            jmp           begin         ;get another character
```

Reset Disk (Function 0DH)



Call
AH = 0DH

Return
None

Function 0DH flushes all file buffers to ensure that the internal buffer cache matches the disks in the drives. It writes out buffers that have been modified, and marks all buffers in the internal cache as free. This function request is normally used to force a known state of the system; Control-C interrupt handlers should call this function.

This function request does not update directory entries; you must close files that have changed to update their directory entries (see Function 10H, Close File).

Macro Definition:

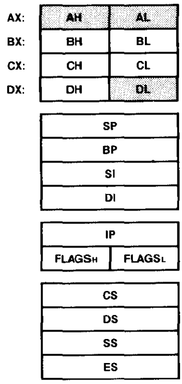
```
reset_disk macro
           mov    ah,0DH
           int    21H
           endm
```

Example

The following program flushes all file buffers and selects disk A.

```
begin:  reset_disk
        select_disk "A"
```

Select Disk (Function 0EH)



Call
AH = 0EH
DL
  Drive number
  (0 = A, 1 = B, etc.)

Return
AL
  Number of logical drives

Function 0EH selects the drive specified in DL (0=A, 1=B, etc.) as the current drive. AL returns the number of drives.

> **Note**
>
> For future compatibility, treat the value returned in AL with care. For example, if AL returns 5, it is not safe to assume drives A, B, C, D, and E are all valid drive designators.

Macro Definition:

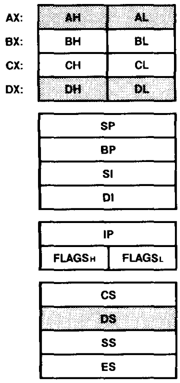
```
select_disk macro disk
            mov   dl,disk[-64]
            mov   ah,0EH
            int   21H
            endm
```

Example

The following program selects the drive not currently selected in a 2-drive system.

```
begin:    current_disk       ;see Function 19H
          cmp    al,00H      ;drive A: selected?
          je     select_b    ;yes, select B
          select_disk "A"    ;THIS FUNCTION
          jmp    return
select_b: select_disk "B"    ;THIS FUNCTION
```

Open File (Function 0FH)



Call
AH = 0FH
DS:DX
  Pointer to unopened FCB

Return
AL
  0 = Directory entry found
  FFH = No directory entry found

Function 0FH opens a file. DX must contain the offset (from the segment address in DS) of an unopened File Control Block (FCB). The disk directory is searched for the named file.

If a directory entry for the file is found, AL returns 0 and the FCB is filled as follows:

> If the drive code was 0 (current drive), it is changed to the actual drive used (1=A, 2=B, etc.). This lets you change the current drive without interfering with subsequent operations on this file.
>
> Current Block (offset 0CH) is set to 0.
>
> Record Size (offset 0EH) is set to the system default of 128.
>
> File Size (offset 10H), Date of Last Write (offset 14H), and Time of Last Write (offset 16H) are set from the directory entry.

Before performing a sequential disk operation on the file, you must set the Current Record field (offset 20H). Before performing a random disk operation on the file, you must set the Relative Record field (offset 21H). If the default record size (128 bytes) is not correct, set it to the correct length.

If a directory entry for the file is not found, or if the file has the hidden or system attribute, AL returns FFH.

Macro Definition:

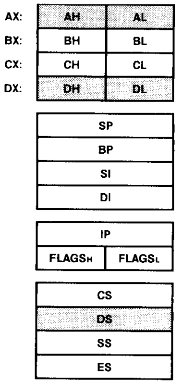
```
open macro fcb
     mov   dx,offset fcb
     mov   ah,0FH
     int   21H
     endm
```

Example

The following program prints the file named TEXTFILE.ASC that is on the disk in drive B. If a partial record is in the buffer at end-of-file, the routine that prints the partial record prints characters until it encounters an end-of-file mark (ASCII 26, or Control-Z).

```
fcb             db     2,"TEXTFILEASC"
                db     26 dup (?)
buffer          db     128 dup (?)
;
begin:          set_dta  buffer         ;see Function 1AH
                open  fcb               ;THIS FUNCTION
read_line:      read_seq  fcb           ;see Function 14H
                cmp   al,02H            ;end of file?
                je    all_done          ;yes, go home
                cmp   al,00H            ;more to come?
                jg    check_more        ;no, check for partial
                                        ;record
                mov   cx,80H            ;yes, print the buffer
                xor   si,si             ;set index to 0
print_it:       print_char buffer[si]   ;see Function 05H
                inc   si                ;bump index
                loop  print_it          ;print next character
                jmp   read_line         ;read another record
check_more:     cmp   al,03H            ;part. record to print?
                jne   all_done          ;no
                mov   cx,80H            ;yes, print it
                xor   si,si             ;set index to 0
find_eof:       cmp   buffer[si],26     ;end-of-file mark?
                je    all_done          ;yes
                print_char buffer[si]   ;see Function 05H
                inc   si                ;bump index to next
                                        ;character
                loop  find_eof
all_done:       close fcb               ;see Function 10H
```

Close File (Function 10H)



Call
AH = 10H
DS:DX
  Pointer to opened FCB

Return
AL
  0 = Directory entry found
  FFH = No directory entry found

Function 10H closes a file. DX must contain the offset (to the segment address in DS) of an opened FCB. The disk directory is searched for the file named in the FCB. If a directory entry for the file is found, the location of the file is compared with the corresponding entries in the FCB. The directory entry is updated, if necessary, to match the FCB, and AL returns 0.

This function must be called after a file is changed to update the directory entry. It is strongly advised that any FCB (even one for a file that hasn't been changed) be closed when access to the file is no longer needed.

If a directory entry for the file is not found, AL returns FFH.

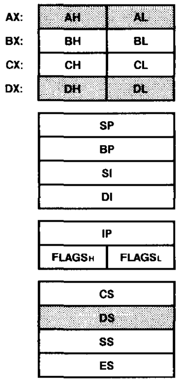
Macro Definition:
```
close macro fcb
      mov   dx,offset fcb
      mov   ah,10H
      int   21H
      endm
```

Example

The following program checks the first byte of the file named MOD1.BAS in drive B to see if it is FFH, and prints a message if it is.

```
message         db    "Not saved in ASCII format",0DH,0AH,"$"
fcb             db     2,"MOD1    BAS"
                db     26  dup (?)
buffer          db     128 dup (?)
;
begin:          set_dta  buffer         ;see Function 1AH
                open   fcb              ;see Function 0FH
                read_seq  fcb           ;see Function 14H
                cmp    buffer,0FFH      ;is first byte FFH?
                jne    all_done         ;no
                display  message        ;see Function 09H
all_done:       close fcb               ;THIS FUNCTION
```

Search for First Entry (Function 11H)



Call
AH = 11H
DS:DX
  Pointer to unopened FCB

Return
  AL
  0 = Directory entry found
  FFH = No directory entry found

Function 11H searches the disk directory for the first matching filename. DX must contain the offset (from the segment address in DS) of an unopened FCB. The filename in the FCB can include wildcard characters. To search for hidden or system files, DX must point to the first byte of an extended FCB prefix.

If a directory entry for the filename in the FCB is not found, AL returns FFH.

If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address as follows:

If the search FCB was normal, the first byte at the Disk Transfer Address is set to the drive number used (1=A, 2=B, etc.) and the next 32 bytes contain the directory entry.

If the search FCB was extended, the first byte at the Disk Transfer Address is set to FFH, the next 5 bytes are set to 00H, and the following byte is set to the value of the attribute byte in the search FCB. The remaining 33 bytes are the same as the result of the normal FCB (drive number and 32 bytes of directory entry).

If Function 12H (Search for Next Entry) is used to continue searching for matching filenames, the original FCB at DS:DX must not be altered or opened.

The attribute field is the last byte of the extended FCB fields that precede the FCB (see "Extended FCB" in Section 1.8.1 File Control Block (FFCB)). If the attribute field is zero, only normal file entries are searched. Directory entries for hidden files, system files, volume label, and subdirectories are not searched.

If the attribute field is hidden file, system file, or directory entry (02H, 04H, or 10H), or any combination of those values, all normal file entries are also searched. To search all directory entries except the volume label, set the attribute byte to 16H (hidden file and system file and directory entry).

If the attribute field is volume label (08H), only the volume label entry is searched.

Macro Definition:

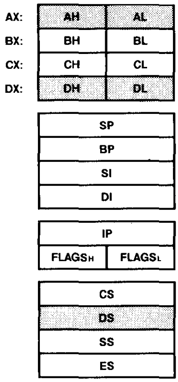
```
search_first macro fcb
             mov   dx,offset fcb
             mov   ah,11H
             int   21H
             endm
```

Example

The following program verifies the existence of a file named REPORT.ASM on the disk in drive B.

```
yes         db    "FILE EXISTS.$"
no          db    "FILE DOES NOT EXIST.$"
crlf        db     0DH,0AH,"$"
fcb         db     2,"REPORT  ASM"
            db     26 dup (?)
buffer      db     128 dup (?)
;
begin:      set_dta   buffer              ;see Function 1AH
            search_first fcb              ;THIS FUNCTION
            cmp       al,0FFH             ;directory entry found?
            je        not_there           ;no
            display   yes                 ;see Function 09H
            jmp       continue
not_there:  display   no                  ;see Function 09H
continue:   display   crlf                ;see Function 09H
```

Search for Next Entry (Function 12H)



Call
AH = 12H
DS:DX
  Pointer to unopened FCB

Return
AL
  0 = Directory entry found
  FFH = No directory entry found

Function 12H is used after Function 11H (Search for First Entry) to find additional directory entries that match a filename that contains wildcard characters. It searches the disk directory for the next matching name. DX must contain the offset (from the segment address in DS) of an FCB previously specified in a call to Function 11H. To search for hidden or system files, DX must point to the first byte of an extended FCB prefix that includes the appropriate attribute value.

If a directory entry for the filename in the FCB is not found, AL returns FFH.

If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address (see Function 11H for a description of how the unopened FCB is formed).

Macro Definition:

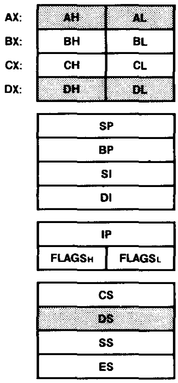
```
search_next macro fcb
            mov   dx,offset fcb
            mov   ah,12H
            int   21H
            endm
```

Example

The following program displays the number of files on the disk in drive B.

```
message      db    "No files",0DH,0AH,"$"
files        db    0
fcb          db    2,"???????????"
             db    26  dup (?)
buffer       db    128 dup (?)
;
begin:       set_dta  buffer          ;see Function 1AH
             search_first  fcb        ;see Function 11H
             cmp  al,0FFH             ;directory entry found?
             je   all_done            ;no, no files on disk
             inc  files               ;yes, increment file
                                      ;counter
search_dir:  search_next  fcb         ;THIS FUNCTION
             cmp  al,0FFH             ;directory entry found?
             je   done                ;no
             inc  files               ;yes, increment file
                                      ;counter
             jmp  search_dir          ;check again
done:        convert files,10,message ;see end of chapter
all_done:    display  message         ;see Function 09H
```

Delete File (Function 13H)



Call
AH = 13H
DS:DX
  Pointer to unopened FCB

Return
  AL
  0 = Directory entry found
  FFH = No directory entry found

Function 13H deletes a file. DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for a matching filename. The filename in the FCB can contain wildcard characters.

If no matching directory entry is found, AL returns FFH.

If a matching directory entry is found, AL returns 0 and the entry is deleted from the directory. If a wildcard character is used in the filename, all files which match will be deleted.

Do not delete open files.

Macro Definition:

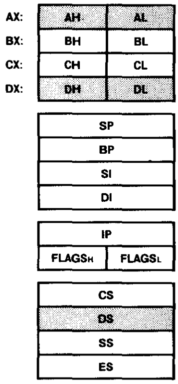
```
delete  macro   fcb
        mov     dx,offset fcb
        mov     ah,13H
        int     21H
        endm
```

Example

The following program deletes each file on the disk in drive B that was last written before December 31, 1982.

```
year            dw      1982
month           db      12
day             db      31
files           db      0
message         db      "NO FILES DELETED.",0DH,0AH,"$"
fcb             db      2,"???????????"
                db      26 dup  (?)
buffer          db      128 dup (?)
;
begin:          set_dta  buffer        ;see Function 1AH
                search_first fcb       ;see Function 11H
                cmp     al,0FFH        ;directory entry found?
                jne     compare        ;yes
                jmp     all_done       ;no, no files on disk
compare:        convert_date buffer    ;see end of chapter
                cmp    cx,year         ;next several lines
                jg     next            ;check date in directory
                cmp    dl,month        ;entry against date
                jg     next            ;above & check next file
                cmp    dh,day          ;if date in directory
                jge    next            ;entry isn't earlier.
                delete  buffer         ;THIS FUNCTION
                inc    files           ;bump deleted-files
                                       ;counter
next:           search_next fcb        ;see Function 12H
                cmp    al,00H          ;directory entry found?
                je     compare         ;yes, check date
                cmp    files,0         ;any files deleted?
                je     all_done        ;no, display NO FILES
                                       ;message.
                convert files,10,message ;see end of chapter
all_done:       display message        ;see Function 09H
```

Sequential Read (Function 14H)



Call
AH = 14H
DS:DX
  Pointer to opened FCB

Return
AL
  00H = Read completed successfully
  01H = EOF
  02H = DTA too small
  03H = EOF, partial record

Function 14H reads a record from the specified file. DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by the Current Block field (offset 0CH) and Current Record (offset 20H) field is loaded at the Disk Transfer Address, then the Current Block and Current Record fields are incremented.

The length of the record is taken from the Record Size field (offset 0EH) of the FCB.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully.
1	End-of-file; no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

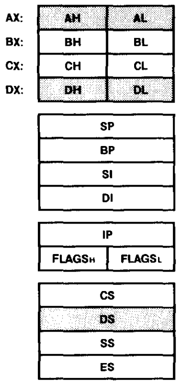
Macro Definition:
```
read_seq macro fcb
         mov   dx,offset fcb
         mov   ah,14H
         int   21H
         endm
```

Example

The following program displays the file named TEXTFILE.ASC that is on the disk in drive B; its function is similar to the MS-DOS Type command. If a partial record is in the buffer at end of file, the routine that displays the partial record displays characters until it encounters an end-of-file mark (ASCII 1AH, or Control-Z).

```
fcb             db    2,"TEXTFILEASC"
                db    26  dup (?)
buffer          db    128 dup (?),"$"
;
begin:          set_dta  buffer      ;see Function 1AH
                open  fcb            ;see Function 0FH
read_line:      read_seq  fcb        ;THIS FUNCTION
                cmp   al,02H         ;DTA too small?
                je    all_done       ;yes
                cmp   al,00H         ;end-of-file?
                jg    check_more     ;yes
                display  buffer      ;see Function 09H
                jmp   read_line      ;get another record
check_more:     cmp   al,03H         ;partial record in buffer?
                jne   all_done       ;no, go home
                xor   si,si          ;set index to 0
find_eof:       cmp   buffer[si],26 ;is character EOF?
                je    all_done       ;yes, no more to display
                display_char buffer[si]  ;see Function 02H
                inc   si             ;bump index
                jmp   find_eof       ;check next character
all_done:       close fcb            ;see Function 10H
```

Sequential Write (Function 15H)



Call
AH = 15H
DS:DX
  Pointer to opened FCB

Return
AL
  00H = Write completed successfully
  01H = Disk full
  02H = DTA too small

Function 15H writes a record to the specified file. DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by Current Block field (offset 0CH) and Current Record field (offset 20H) is written from the Disk Transfer Address, then the Current Block and Current Record fields are incremented.

The record size is taken from the value of the Record Size field (offset 0EH) of the FCB. If the Record Size is less than a sector, the data at the Disk Transfer Address is written to an MS-DOS buffer; MS-DOS writes the buffer to disk when it contains a full sector of data, or the file is closed, or a Reset Disk system call (Function 0DH) is issued.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully.
1	Disk full; write canceled.
2	Not enough room at the Disk Transfer Address to write one record; write canceled.

Macro Definition: write_seq macro fcb

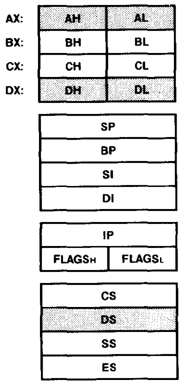
```
                          mov   dx,offset fcb
                          mov   ah,15H
                          int   21H
                          endm
```

Example

The following program creates a file named DIR.TMP on the disk in drive B that contains the disk number (0=A, 1=B, etc.) and filename from each directory entry on the disk.

```
record_size     equ    0EH                 ;offset of Record Size
;                                          field in FCB
fcbl            db     2,"DIR      TMP"
                db     26 dup (?)
fcb2            db     2,"???????????"
                db     26 dup (?)
buffer          db     128 dup (?)
;
begin:        set_dta       buffer        ;see Function 1AH
              search_first  fcb2          ;see Function 11H
              cmp           al,0FFH       ;directory entry found?
              je            all_done      ;no, no files on disk
              create        fcbl          ;see Function 16H
              mov           fcbl[record_size],12
                                          ;set record size to 12
write_it:     write_seq     fcbl          ;THIS FUNCTION
              cmp           al,0          ;write successful?
              jne           all_done      ;no, go home
              search_next   fcb2          ;see Function 12H
              cmp           al,FFH        ;directory entry found?
              je            all_done      ;no, go home
              jmp           write_it      ;yes, write the record
all_done:     close         fcbl          ;see Function 10H
```

Create File (Function 16H)



Call
AH = 16H
DS:DX
  Pointer to unopened FCB

Return
AL
  00H = Empty directory found
  FFH = No empty directory
        available

Function 16H creates a file. DX must contain the offset (from the segment address in DS) of an unopened FCB. MS-DOS searches the directory for an entry that matches the specified filename or, if there is no matching entry, an empty entry.

If MS-DOS finds a matching entry, it opens the file and sets the length to zero (in other words, if you try to create a file that already exists, MS-DOS erases it and creates a new, empty file). If MS-DOS doesn't find a matching entry but does find an empty directory entry, it opens the file and sets its length to zero. In either case, the file is created and AL returns 0. If MS-DOS doesn't find a matching entry and there is no empty entry, the file is not created and AL returns FFH.

You can assign an attribute to the file by using an extended FCB with the attribute byte set to the appropriate value (see "Extended FCB" in Section 1.8.1).

Macro Definition:

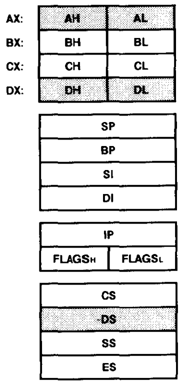
```
create macro fcb
       mov   dx,offset fcb
       mov   ah,16H
       int   21H
       endm
```

Example

The following program creates a file named DIR.TMP on the disk in drive B that contains the disk number (0 = A, 1 = B, etc.) and filename from each directory entry on the disk.

```
record_size     equ    0EH              ;offset of Record Size
;                                        field of FCB
fcb1            db     2,"DIR     TMP"
                db     26  dup (?)
fcb2            db     2,"???????????"
                db     26  dup (?)
buffer          db     128 dup (?)
;
begin:          set_dta    buffer        ;see Function 1AH
                search_first  fcb2      ;see Function 11H
                cmp        al,0FFH       ;directory entry found?
                je         all_done      ;no, no files on disk
                create     fcb1          ;THIS FUNCTION
                mov        fcb1[record_size],12
                                         ;set record size to 12
write_it:       write_seq fcb1           ;see Function 15H
                cmp        al,0          ;write successful
                jne        all_done      ;no, go home
                search_next  fcb2        ;see Function 12H
                cmp        al,FFH        ;directory entry found?
                je         all_done      ;no, go home
                jmp        write_it      ;yes, write the record
all_done:       close      fcb1          ;see Function 10H
```

Rename File (Function 17H)



Call
AH = 17H
DS:DX
  Pointer to modified FCB

Return
AL
  00H = Directory entry found
  FFH = No directory entry found or destination already exists

Function 17H changes the name of an existing file. DX must contain the offset (from the segment address in DS) of an FCB with the drive number and filename filled in, followed by a second filename at offset 11H. DOS searches the disk directory for an entry that matches the first filename, which can contain wildcard characters.

If MS-DOS finds a matching directory entry and there is no directory entry that matches the second filename, it changes the filename in the directory entry to match the second filename in the modified FCB and AL returns zero. If a wildcard character is used in the second filename, the corresponding characters in the filename of the directory entry are not changed.

This function request cannot be used to rename a hidden file, a system file, or a subdirectory. If MS-DOS does not find a matching directory entry or finds an entry for the second filename, AL returns FFH.

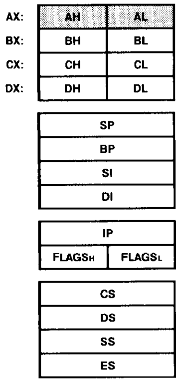
Macro Definition:
```
rename macro fcb,newname
       mov   dx,offset fcb
       mov   ah,17H
       int   21H
       endm
```

Example

The following program prompts for the name of a file and a new name, then renames the file.

```
fcb             db    37 dup (?)
prompt1         db    "Filename: $"
prompt2         db    "New name: $"
reply           db    15 dup(?)
crlf            db    0DH,0AH,"$"
;
begin:          display  prompt1        ;see Function 09H
                get_string  15,reply    ;see Function 0AH
                display  crlf           ;see Function 09H
                parse    reply[2],fcb   ;see Function 29H
                display  prompt2        ;see Function 09H
                get_string  15,reply    ;see Function 0AH
                display  crlf           ;see Function 09H
                parse    reply[2],fcb[16]
                                        ;see Function 29H
                rename   fcb            ;THIS FUNCTION
```

Get Current Disk (Function 19H)



Call
AH = 19H

Return
AL
Currently selected drive
(0 = A, 1 = B, etc.)

Function 19H returns the current drive in AL (0=A, 1=B, etc.).

Macro Definition:

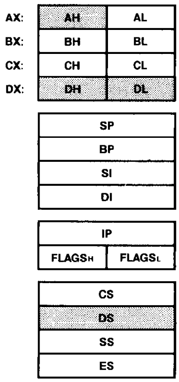
```
current_disk  macro
              mov   ah,19H
              int   21H
              endm
```

Example

The following program displays the currently selected (default) drive in a 2-drive system.

```
message     db   "Current disk is $"
crlf        db   0DH,0AH,"$"
;
begin:      display  message          ;see Function 09H
            current_disk              ;THIS FUNCTION
            cmp      al,00H           ;is it disk A?
            jne      disk_b           ;no, it's disk B:
            display_char "A"          ;see Function 02H
            jmp      all_done
disk_b:     display_char "B"          ;see Function 02H
all_done:   display  crlf             ;see Function 09H
```

Set Disk Transfer Address (Function 1AH)



Call
AH = 1AH
DS:DX
 Disk Transfer Address

Return
None

Function 1AH sets the Disk Transfer Address. DX must contain the offset (from the segment address in DS) of the Disk Transfer Address. Disk transfers cannot wrap around from the end of the segment to the beginning, nor can they overflow into another segment.

If you do not set the Disk Transfer Address, MS-DOS defaults to offset 80H in the Program Segment Prefix. You can check the current Disk Transfer Address with Function 2FH (Get Data Transfer Address).

Macro Definition:

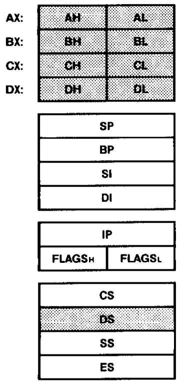
```
set_dta  macro  buffer
         mov    dx,offset buffer
         mov    ah,1AH
         int    21H
```

Example

The following program prompts for a letter, converts the letter to its alphabetic sequence (A=1, B=2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B. The file contains 26 records; each record is 28 bytes long.

```
record_size     equ   0EH      ;offset of Record Size
                               ;field of FCB
relative_record equ   21H      ;offset of Relative Record
;                              field of FCB
fcb           db    2,"ALPHABETDAT"
              db    26  dup (?)
buffer        db    28 dup(?),"$"
prompt        db   "Enter letter: $"
crlf          db    0DH,0AH,"$"
;
begin:        set_dta  buffer        ;THIS FUNCTION
              open     fcb           ;see Function 0FH
              mov      fcb[record_size],28 ;set record size
get_char:     display  prompt        ;see Function 09H
              read_kbd_and_echo      ;see Function 01H
              cmp      al,0DH        ;just a CR?
              je       all_done      ;yes, go home
              sub      al,41H        ;convert ASCII
                                     ;code to record #
              mov      fcb[relative_record],al
                                     ;set relative record
              display  crlf          ;see Function 09H
              read_ran fcb           ;see Function 21H
              display  buffer        ;see Function 09H
              display  crlf          ;see Function 09H
              jmp      get_char      ;get another character
all_done:     close    fcb           ;see Function 10H
```

Get Default Drive Data (Function 1BH)



Call
AH = 1BH

Return
AL
  Sectors per cluster
CX
  Bytes per sector
DX
  Clusters per drive
DS:BX
  Pointer to FAT ID byte

Function 1BH retrieves data about the disk in the default drive. The data is returned in the following registers:

AL	The number of sectors in a cluster (allocation unit).
CX	The number of bytes in a sector.
DX	The number of clusters on the disk.

BX returns the offset (to the segment address in DS) of the first byte of the File Allocation Table (FAT), which identifies the type of disk in the drive:

Value	Type of Drive
FF	Double-sided diskette, 8 sectors per track.
FE	Single-sided diskette, 8 sectors per track.
FD	Double-sided diskette, 9 sectors per track.
FC	Single-sided diskette, 9 sectors per track.
F9	Double-sided diskette, 15 sectors per track.
F8	Fixed disk.

This call is similar to Function 36H (Get Disk Free Space), except that it returns the address of the FAT ID byte in BX instead of the number of available clusters, and to Function 1CH (Get Drive Data), except that it returns data on the disk in the default drive instead of the disk in a specified drive. For a description of how MS-DOS stores data on a disk, including a description of the File Allocation Table, see Chapter 3.

Macro Definition: def_drive_data

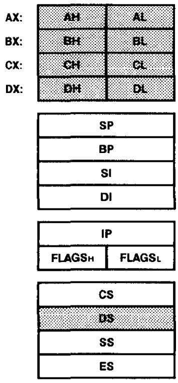
```
def_drive_data  macro
                push    ds
                mov     ah,1BH
                int     21H
                mov     al,byte ptr[bx]
                pop     ds
                endm
```

Example

The following program displays a message that tells whether the default drive is a diskette or fixed disk drive.

```
stdout      equu        1
;
msg         db          "Default drive is "
remov       db          "diskette."
fixed       db          "fixed."
crlf        db          ODH,OAH
;
begin:      write_handle stdout,msg,17      ;display message
            jc           write_error        ;routine not shown
            def_drive_data                  ;THIS FUNCTION
            cmp          byte ptr [bx],0F8H ;check FAT ID byte
            jne          diskette           ;it's a diskette
            write_handle stdout,fixed,6     ;see Function 40H
            jc           write_error        ;see Function 40H
            jmp short    all_done           ;clean up & go home
diskette:   write_handle stdout,remov,9     ;see Function 40H
all_done:   write_handle stdout,crlf,2      ;see Function 40H
            jc           write_error        ;routine not shown
```

Get Drive Data (Function 1CH)



Call
AH = 1CH
DL
   Drive (0=default, 1=A, etc.)

Return
AL
   0FFH if drive number is invalid,
   otherwise sectors per cluster
CX
   Bytes per sector
DX
   Clusters per drive
DS:BX
   Pointer to FAT ID byte

Function 1CH retrieves data about the disk in the specified drive. DL must contain the drive number (0=default, 1=A, etc.). The data is returned in the following registers:

AL	The number of sectors in a cluster (allocation unit).
CX	The number of bytes in a sector.
DX	The number of clusters on the disk.

BX returns the offset (to the segment address in DS) of the first byte of the File Allocation Table (FAT), which identifies the type of disk in the drive:

Value	Type of Drive
FF	Double-sided diskette, 8 sectors per track.
FE	Single-sided diskette, 8 sectors per track.
FD	Double-sided diskette, 9 sectors per track.
FC	Single-sided diskette, 9 sectors per track.
F9	Double-sided diskette, 15 sectors per track.
F8	Fixed disk.

If the drive number in DL is invalid, AL returns 0FFH.

This call is similar to Function 36H (Get Disk Free Space), except that it returns the address of the FAT ID byte in BX instead of the number of available clusters, and to Function 1BH (Get Default Drive Data), except that it returns data on the disk in the drive specified in DL instead of the disk in the default drive. For a description of how MS-DOS stores data on a disk, including a description of the File Allocation Table, see Chapter 3.

Macro Definition:

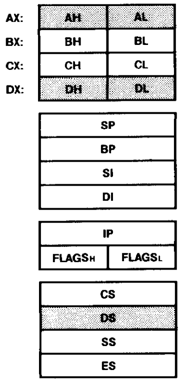
```
drive_data  macro   drive
            push    ds
            mov     dl,drive
            mov     ah,1BH
            int     21H
            mov     al, byte ptr[bx]
            pop     ds
            endm
```

Example

The following program displays a message that tells whether drive B is a diskette or fixed disk drive.

```
stdout      equu          1
:
msg         db            "Drive B is "
remov       db            "diskette."
fixed       db            "fixed."
crlf        db            ODH,OAH
;
begin:      write_handle  stdout,msg,11        ;display message
            jc            write_error          ;routine not shown
            drive_data    2                    ;THIS FUNCTION
            cmp           byte ptr [bx],0F8H   ;check FAT ID byte
            jne           diskette             ;it's a diskette
            write_handle  stdout,fixed,6       ;see Function 40H
            jc            write_error          ;routine not shown
            jmp           all_done             ;clean up & go home
diskette:   write_handle  stdout,remov,9       ;see Function 40H
all_done:   write_handle  stdout,crlf,2        ;see Function 40H
            jc            write_error          ;routine not shown
```

Random Read (Function 21H)



Call
AH = 21H
DS:DX
  Pointer to opened FCB

Return
AL
  0 = Read completed successfully
  1 = End of file, record empty
  2 = DTA too small
  3 = End of file, partial record

Function 21H reads the record pointed to by the Relative Record field (offset 21H) of the FCB to the Disk Transfer Address. DX must contain the offset (from the segment address in DS) of an opened FCB. The Current Block field (offset 0CH) and Current Record field (offset 20H) are set to agree with the Relative Record field (offset 21H), then the record is loaded at the Disk Transfer Address. The record length is taken from the Record Size field (offset 0EH) of the FCB.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully.
1	End-of-file; no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

Macro Definition:

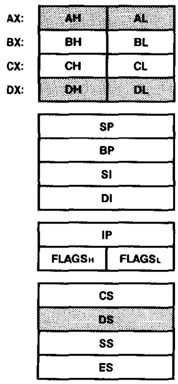
```
read_ran  macro   fcb
          mov     dx,offset fcb
          mov     ah,21H
          int     21H
          endm
```

Example

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B. The file contains 26 records; each record is 28 bytes long.

```
record_size      equu   0EH        ;offset of Record Size
                                   ;field of FCB
relative_record equu   21H        ;offset of Relative Record
;                                  field of FCB
fcb           db    2,"ALPHABETDAT"
              db    26  dup (?)
buffer        db    28 dup(?),"$"
prompt        db   "Enter letter: $"
crlf          db    0DH,0AH,"$"
;
begin:        set_dta  buffer              ;see Function 1AH
              open     fcb                 ;see Function 0FH
              mov      fcb[record_size],28 ;set record size
get_char:     display  prompt              ;see Function 09H
              read_kbd_and_echo            ;see Function 01H
              cmp      al,0DH              ;just a CR?
              je       all_done            ;yes, go home
              sub      al,41H              ;convert ASCII code
                                           ;to record #
              mov      fcb[relative_record],al ;set relative
                                           ;record
              display  crlf                ;see Function 09H
              read_ran fcb                 ;THIS FUNCTION
              display  buffer              ;see Function 09H
              display  crlf                ;see Function 09H
              jmp      get_char            ;get another char.
all_done:     close    fcb                 ;see Function 10H
```

Random Write (Function 22H)



Call
AH = 22H
DS:DX
  Pointer to opened FCB

Return
AL
  00H = Write completed successfully
  01H = Disk full
  02H = DTA too small

Function 22H writes the record pointed to by the Relative Record field (offset 21H) of the FCB from the Disk Transfer Address. DX must contain the offset from the segment address in DS of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is written from the Disk Transfer Address.

The record length is taken from the Record Size field (offset 0EH) of the FCB. If the record size is less than a sector, the data at the Disk Transfer Address is written to a buffer; the buffer is written to disk when it contains a full sector of data, or the file is closed, or a Reset Disk system call (Function 0DH) is issued.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully.
1	Disk is full.
2	Not enough room at the Disk Transfer Address to write one record; write canceled.

Macro Definition:

```
write_ran  macro   fcb
           mov     dx,offset fcb
           mov     ah,22H
           int     21H
           endm
```

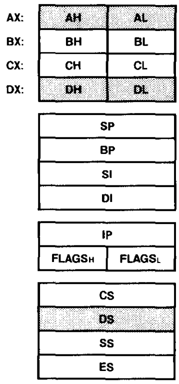
Example

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B. After displaying the record, it prompts the user to enter a changed record. If the user types a new record, it is written to the file; if the user just presses Return, the record is not replaced. The file contains 26 records; each record is 28 bytes long.

```
record_size     equ    0EH        ;offset of Record Size
                                  ;field of FCB
relative_record equ    21H        ;offset of Relative Record
;                                 field of FCB
fcb         db    2,"ALPHABETDAT"
            db    26  dup (?)
buffer      db    28 dup(?),0DH,0AH,"$"
prompt1     db   "Enter letter: $"
prompt2     db   "New record (RETURN for no change): $"
crlf        db    0DH,0AH,"$"
reply       db    28 dup (32)
blanks      db    26 dup (32)
;
begin:      set_dta  buffer               ;see Function 1AH
            open     fcb                  ;see Function 0FH
            mov      fcb[record_size],28 ;set record size
get_char:   display  prompt1              ;see Function 09H
            read_kbd_and_echo             ;see Function 01H
            cmp      al,0DH               ;just a CR?
            je       all_done             ;yes, go home
            sub      al,41H               ;convert ASCII
                                          ;code to record #
            mov      fcb[relative_record],al
                                          ;set relative record
            display  crlf                 ;see Function 09H
            read_ran fcb                  ;THIS FUNCTION
            display  buffer               ;see Function 09H
            display  crlf                 ;see Function 09H
            display  prompt2              ;see Function 09H
            get_string 27,reply           ;see Function 0AH
            display  crlf                 ;see Function 09H
            cmp      reply[1],0           ;was anything typed
                                          ;besides CR?
            je       get_char             ;no
                                          ;get another char.
            xor      bx,bx                ;to load a byte
            mov      bl,reply[1]          ;use reply length as
                                          ;counter
```

```
          move_string blanks,buffer,26 ;see chapter end
          move_string reply[2],buffer,bx ;see chapter end
          write_ran fcb                ;THIS FUNCTION
          jmp     get_char             ;get another character
all_done: close   fcb                  ;see Function 10H
```

Get File Size (Function 23H)



Call
AH = 23H
DS:DX
  Pointer to unopened FCB

Return
AL
  00H = Directory entry found
  FFH = No directory entry found

Function 23H returns the size of the specified file. DX must contain the offset (from the segment address in DS) of an unopened FCB.

If there is a directory entry that matches the specified file, MS-DOS divides the File Size field (offset 1CH) of the directory entry by the Record Size field (offset 0EH) of the FCB, puts the result in the Relative Record field (offset 21H) of the FCB, and returns 00 in AL.

You must set the Record Size field of the FCB to the correct value before calling this function. If the Record Size field is not an even divisor of the File Size field, the value set in the Relative Record field is rounded up, yielding a value larger than the actual number of records.

If no matching directory is found, AL returns FFH.

Macro Definition:

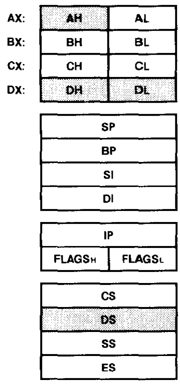
```
file_size  macro  fcb
           mov    dx,offset fcb
           mov    ah,23H
           int    21H
           endm
```

Example

The following program prompts for the name of a file, opens the file to fill in the Record Size field of the FCB, issues a File Size system call, and displays the record length and number of records.

```
fcb             db      37 dup  (?)
prompt          db      "File name: $"
msg1            db      "Record length:     ",0DH,0AH,"$"
msg2            db      "Records:      ",0DH,0AH,"$"
crlf            db      0DH,0AH,"$"
reply           db      17 dup(?)
;
begin:          display prompt             ;see Function 09H
                get_string 17,reply        ;see Function 0AH
                cmp     reply[1],0         ;just a CR?
                jne     get_length         ;no, keep going
                jmp     all_done           ;yes, go home
get_length:     display crlf               ;see Function 09H
                parse   reply[2],fcb       ;see Function 29H
                open    fcb                ;see Function 0FH
                file_size fcb              ;THIS FUNCTION
                mov     ax,word ptr fcb[33] ;get record length
                convert ax,10,msg2[9]      ;see end of chapter
                mov     ax,word ptr fcb[14] ; get record number
                convert ax,10,msg1[15]     ;see end of chapter
                display msg1               ;see Function 09H
                display msg2               ;see Function 09H
all_done:       close   fcb                ;see Function 10H
```

Set Relative Record (Function 24H)



Call
AH = 24H
DS:DX
  Pointer to opened FCB

Return
None

Function 24H sets the Relative Record field (offset 21H) to the file address specified by the Current Block field (offset 0CH) and Current Record field (offset 20H). DX must contain the offset (from the segment address in DS) of an opened FCB. You use this call to set the file pointer before a random read or write (Functions 21H, 22H, 27H, or 28H).

Macro Definition:

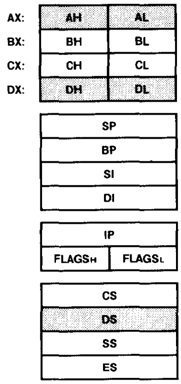
```
set_relative_record  macro   fcb
                     mov     dx,offset fcb
                     mov     ah,24H
                     int     21H
                     endm
```

Example

The following program copies a file using the Random Block Read and Random Block Write system calls. It speeds the copy by setting the record length equal to the file size and the record count to 1, and using a buffer of 32K bytes. It positions the file pointer by setting the Current Record field (offset 20H) to 1 and using Set Relative Record to make the Relative Record field (offset 21H) point to the same record as the combination of the Current Block field (offset 0CH) and Current Record field (offset 20H).

```
current_record  equ    20H          ;offset of Current Record
                                    ;field of FCB
fil_size        equu   10H          ;offset of File Size
;                                   field of FCB
fcb        db      37 dup (?)
filename   db      17 dup(?)
prompt1    db     "File to copy: $"  ;see Function 09H for
prompt2    db     "Name of copy: $"  ;explanation of $
crlf       db      0DH,0AH,"$"
file_length  dw    ?
buffer     db      32767 dup(?)
;
begin:     set_dta  buffer                 ;see Function 1AH
           display  prompt1                ;see Function 09H
           get_string  15,filename         ;see Function 0AH
           display  crlf                   ;see Function 09H
           parse    filename[2],fcb        ;see Function 29H
           open     fcb                    ;see Function 0FH
           mov      fcb[current_record],0 ;set Current Record
                                           ;field
           set_relative_record  fcb        ;THIS FUNCTION
           mov      ax,word ptr fcb[fil_size] ;get file size
           mov      file_length,ax         ;save it for
                                           ;ran_block_write
           ran_block_read  fcb,1,ax        ;see Function 27H
           display  prompt2                ;see Function 09H
           get_string  15,filename         ;see Function 0AH
           display  crlf                   ;see Function 09H
           parse    filename[2],fcb        ;see Function 29H
           create   fcb                    ;see Function 16H
           mov      fcb[current_record],0 ;set Current Record
                                           ;field
           set_relative_record  fcb        ;THIS FUNCTION
           mov      ax,file_length         ;get original file
                                           ;.length
           ran_block_write  fcb,1,ax       ;see Function 28H
           close    fcb                    ;see Function 10H
```

Set Interrupt Vector (Function 25H)



Call
AH = 25H
AL
  Interrupt number
DS:DX
  Pointer to interrupt-handling routine

Return
None

Function 25H sets the address in the interrupt vector table for the specified interrupt.

AL must contain the number of the interrupt. DX must contain the offset (to the segment address in DS) of the interrupt-handling routine.

To avoid compatibility problems, programs should never read an interrupt vector directly from memory, nor set an interrupt vector by writing it into memory. Use Function 35H (Get Interrupt Vector) to get a vector and this function request to set a vector, unless it is absolutely imperative that your program be compatible with pre-2.0 versions of MS-DOS.

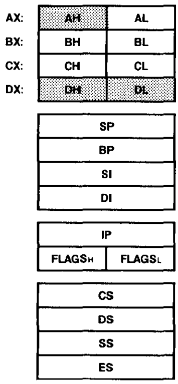
Macro Definition:

```
set_vector   macro  interrupt,handler_start
             mov    al,interrupt
             mov    dx,offset handler_start
             mov    ah,25H
             endm
```

Example

Because interrupts tend to be machine-specific, no example is shown.

Create New PSP (Function 26H)



Call
AH = 26H
DX
  Segment address of new PSP

Return
None

Function 26H creates a new Program Segment Prefix. DX must contain the segment address where the new PSP is to be created.

This function request has been superseded. Use Function 4BH, Code 0 (Load and Execute Program) to execute a child process unless it is imperative that your program be compatible with pre-2.0 versions of MS-DOS.

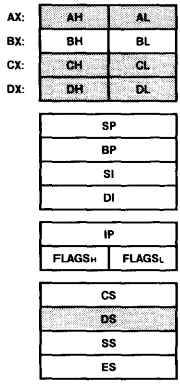
Macro Definition:

```
create_psp  macro   seg_addr
            mov     dx,seg_addr
            mov     ah,26H
            endm
```

Example

Because Function 4BH, Code 0 (Load and Execute Program) and Code 3 (Load Overlay) have superseded this function request, no example is shown.

Random Block Read (Function 27H)



Call
AH = 27H
DS:DX
  Pointer to opened FCB
CX
  Number of blocks to read

Return
AL
  0 = Read completed successfully
  1 = End of file, empty record
  2 = DTA too small
  3 = End of file, partial record
CX
  Number of blocks read

Function 27H reads one or more records from the specified file to the Disk Transfer Address. DX must contain the offset (to the segment address in DS) of an opened FCB. CX must contain the number of records to read. Reading starts at the record specified by the Relative Record field (offset 21H); you must set this field with Function 24H (Set Relative Record) before calling this function.

DOS calculates the number of bytes to read by multiplying the value in CX by the Record Size field (offset 0EH) of the FCB.

CX returns the number of records read. The Current Block field (offset 0CH), Current Record field (offset 20H), and Relative Record field (offset 21H) are set to address the next record.

If you call this function with CX=0, no records are read.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully.
1	End-of-file; no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

Macro Definition:

```
ran_block_read  macro  fcb,count,rec_size
                mov    dx,offset fcb
                mov    cx,count
                mov    word ptr fcb[14],rec_size
                mov    ah,27H
                int    21H
                endm
```

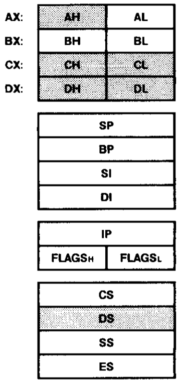
Example

The following program copies a file using the Random Block Read system call. It speeds the copy by specifying a record count of 1 and a record length equal to the file size, and using a buffer of 32K bytes; the file is read as a single record (compare to the sample program for Function 28H that specifies a record <u>length</u> of 1 and a record <u>count</u> equal to the file size).

```
current_record    equ   20H   ;offset of Current Record field
fil_size          equ   10H   ;offset of File Size field
;
fcb         db      37 dup (?)
filename    db      17 dup(?)
prompt1     db     "File to copy: $"    ;see Function 09H for
prompt2     db     "Name of copy: $"    ;explanation of $
crlf        db      0DH,0AH,"$"
file_length  dw     ?
buffer      db      32767 dup(?)
;
begin:     set_dta     buffer                ;see Function 1AH
           display     prompt1               ;see Function 09H
           get_string  15,filename           ;see Function 0AH
           display     crlf                  ;see Function 09H
           parse       filename[2],fcb       ;see Function 29H
           open        fcb                   ;see Function 0FH
           mov         fcb[current_record],0  ;set Current
                                             ;Record field
           set_relative_record fcb           ;see Function 24H
           mov         ax, word ptr fcb[fil_size]
```

```
                                        ;get file size
        mov        file_length,ax       ;save it
        ran_block_read  fcb,1,ax        ;THIS FUNCTION
        display    prompt2              ;see Function 09H
        get_string 15,filename          ;see Function 0AH
        display    crlf                 ;see Function 09H
        parse      filename[2],fcb      ;see Function 29H
        create     fcb                  ;see Function 16H
        mov        fcb[current_record],0;set current
                                        ;Record field
        set_relative_record fcb         ;see Function 24H
        ran_block_write  fcb,1,ax       ;see Function 28H
        close      fcb                  ;see Function 10H
```

Random Block Write (Function 28H)



Call
AH = 28H
DS:DX
  Pointer to opened FCB
CX
  Number of blocks to write
  (0 = set File Size field)

Return
AL
  00H = Write completed successfully
  01H = Disk full
  02H = End of segment
CX
  Number of blocks written

Function 28H writes one or more records to the specified file from the Disk Transfer Address. DX must contain the offset (to the segment address in DS) of an opened FCB; CX must contain either the number of records to write or 0.

If CX is not 0, the specified number of records is written to the file starting at the record specified in the Relative Record field (offset 21H) of the FCB. If CX is 0, no records are written, but MS-DOS sets the File Size field (offset 1CH) of the directory entry to the value in the Relative Record field of the FCB (offset 21H); disk allocation units are allocated or released, as required, to satisfy this new file size.

MS-DOS calculates the number of bytes to write by multiplying the value in CX by the Record Size field (offset 0EH) of the FCB. CX returns the number of records written; the Current Block field (offset 0CH), Current Record field (offset 20H), and Relative Record (offset 21H) field are set to address the next record.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully.
1	Disk full. No records written.
2	Not enough room at the Disk Transfer Address to write one record; write canceled.

Macro Definition:

```
ran_block_write  macro  fcb,count,rec_size
                 mov    dx,offset fcb
                 mov    cx,count
                 mov    word ptr fcb[14],rec_size
                 mov    ah,28H
                 int    21H
                 endm
```

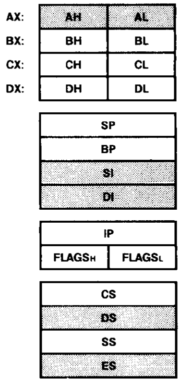
Example

The following program copies a file using the Random Block Read and Random Block Write system calls. It speeds the copy by specifying a record count equal to the file size and a record length of 1, and using a buffer of 32K bytes; the file is copied quickly with one disk access each to read and write (compare to the sample program of Function 27H, that specifies a record <u>count</u> of 1 and a record <u>length</u> equal to file size).

```
current_record  equ  20H    ;offset of Current Record field
fil_size        equ  10H    ;offset of File Size field
;
fcb        db    37 dup (?)
filename   db    17 dup(?)
prompt1    db   "File to copy: $"    ;see Function 09H for
prompt2    db   "Name of copy: $"    ;explanation of $
crlf       db    0DH,0AH,"$"
num_recs   dw    ?
buffer     db    32767 dup(?)
;
begin:     set_dta    buffer        ;see Function 1AH
           display    prompt1       ;see Function 09H
           get_string 15,filename   ;see Function 0AH
           display    crlf          ;see Function 09H
           parse      filename[2],fcb  ;see Function 29H
           open       fcb              ;see Function 0FH
           mov        fcb[current_record],0;set Current
                                       Record field
           set_relative_record fcb     ;see Function 24H
           mov        ax, word ptr fcb[fil_size]
                                       ;get file size
           mov        num_recs,ax      ;save it
```

```
ran_block_read  fcb,num_recs,1  ;THIS FUNCTION
display     prompt2             ;see Function 09H
get_string 15,filename          ;see Function 0AH
display     crlf                ;see Function 09H
parse       filename[2],fcb     ;see Function 29H
create      fcb                 ;see Function 16H
mov         fcb[current_record],0  ;set Current
                                ;Record field
set_relative_record fcb         ;see Function 24H
ran_block_write fcb,num_recs,1 ;see Function 28H
close       fcb                 ;see Function 10H
```

Parse File Name (Function 29H)



Call
AH = 29H
AL
  Controls parsing (see text)
DS:SI
  Pointer to string to parse
ES:DI
  Pointer to buffer for unopened FCB

Return
AL
  00H = No wildcard characters
  01H = Wildcard characters used
  FFH = Drive letter invalid
DS:SI
  Pointer to first byte past string that was parsed
ES:DI
  Pointer to unopened FCB

Function 29H parses a string for a filename of the form drive:filename.extension. SI must contain the offset (to the segment address in DS) of the string to parse; DI must contain the offset (to the segment address in ES) of an area of memory large enough to hold an unopened FCB. If the string contains a valid filename, a corresponding unopened FCB is created at ES:DI.

AL controls the parsing. Bits 4-7 must be 0; bits 0-3 have the following meaning:

Bit	Value	Meaning
0	0	Stop parsing if a file separator is encountered.
	1	Ignore leading separators.
1	0	Set the drive number in the FCB to 0 (current drive) if the string does not contain a drive number.
	1	Leave the drive number in the FCB unchanged if the string does not contain a drive number.
2	0	Set the filename in the FCB to 8 blanks if the string does not contain a filename.

Bit	Value	Meaning
	1	Leave the filename in the FCB unchanged if the string does not contain a filename.
3	1	Leave the extension in the FCB unchanged if the string does not contain an extension.
	0	Set the extension in the FCB to 3 blanks if the string does not contain an extension.

If the string contains a filename or extension that includes an asterisk (*), all remaining characters in the name or extension are set to question mark (?).

Filename separators:

: . ; , = + / " [ ] \ < > | space tab

Filename terminators include all the filename separators plus any control character. A filename cannot contain a filename terminator; if one is encountered, parsing stops.

If the string contains a valid filename:

1. AL returns 1 if the filename or extension contains a wildcard character (* or ?); AL returns 0 if neither the filename nor extension contains a wildcard character.

2. DS:SI points to the first character following the string that was parsed.

   ES:DI points to the first byte of the unopened FCB.

If the drive letter is invalid, AL returns FFH. If the string does not contain a valid filename, ES:DI+1 points to a blank (20H).

Macro Definition:

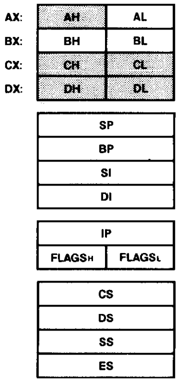
```
parse macro string,fcb
      mov   si,offset string
      mov   di,offset fcb
      push  es
      push  ds
      pop   es
      mov   al,0FH          ;bits 0-3 on
      mov   ah,29H
      int   21H
      pop   es
      endm
```

Example

The following program verifies the existence of the file named in reply to the prompt.

```
fcb             db      37 dup (?)
prompt          db     "Filename: $"
reply           db      17 dup(?)
yes             db     "FILE EXISTS",0DH,0AH,"$"
no              db     "FILE DOES NOT EXIST",0DH,0AH,"$"
                crlf    db 0DH,0AH,"$"
;
begin:          display     prompt          ;see Function 09H
                get_string 15,reply         ;see Function 0AH
                parse       reply[2],fcb    ;THIS FUNCTION
                display     crlf            ;see Function 09H
                search_first  fcb           ;see Function 11H
                cmp         al,0FFH         ;dir. entry found?
                je          not_there       ;no
                display     yes             ;see Function 09H
                jmp         return
not_there:      display     no
```

Get Date (Function 2AH)



Call
AH = 2AH

Return
CX
  Year (1980-2099)
DH
  Month (1-12)
DL
  Day (1-31)
AL
  Day of week (0=Sun., 6=Sat.)

Function 2AH returns the current date set in the operating system as binary numbers in CX and DX:

```
CX  Year (1980-2099)
DH  Month (1=January, 2=February, etc.)
DL  Day (1-31)
AL  Day of week (0=Sunday, 1=Monday, etc.)
```

Macro Definition:

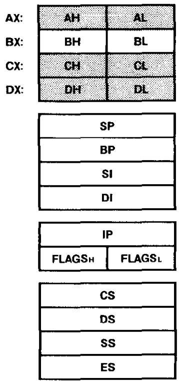
```
get_date  macro
          mov    ah,2AH
          int    21H
          endm
```

Example

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date.

```
month       db      31,28,31,30,31,30,31,31,30,31,30,31
;
begin:      get_date               ;THIS FUNCTION
            inc     dl             ;increment day
            xor     bx,bx          ;so BL can be used as index
            mov     bl,dh          ;move month to index register
            dec     bx             ;month table starts with 0
            cmp     dl,month[bx]   ;past end of month?
            jle     month_ok       ;no, set the new date
            mov     dl,1           ;yes, set day to 1
            inc     dh             ;and increment month
            cmp     dh,12          ;past end of year?
            jle     month_ok       ;no, set the new date
            mov     dh,1           ;yes, set the month to 1
            inc     cx             ;increment year
month_ok:   set_date cx,dh,dl      ;see Function 2AH
```

Set Date (Function 2BH)



Call
AH = 2BH
CX
  Year (1980-2099)
DH
  Month (1-12)
DL
  Day (1-31)

Return
AL
  00H = Date was valid
  FFH = Date was invalid

Function 2BH sets the date in the operating system. Registers CX and DX must contain a valid date in binary:

CX Year (1980-2099)
DH Month (1=January, 2=February, etc.)
DL Day (1-31)

If the date is valid, the date is set and AL returns 0. If the date is not valid, the function is canceled and AL returns FFH.

Macro Definition:

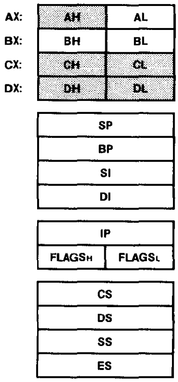
```
set_date  macro  year,month,day
          mov    cx,year
          mov    dh,month
          mov    dl,day
          mov    ah,2BH
          int    21H
          endm
```

Example

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date.

```
month       db      31,28,31,30,31,30,31,31,30,31,30,31
;
begin:      get_date              ;see Function 2AH
            inc    dl             ;increment day
            xor    bx,bx          ;so BL can be used as index
            mov    bl,dh          ;move month to index register
            dec    bx             ;month table starts with 0
            cmp    dl,month[bx]   ;past end of month?
            jle    month_ok       ;no, set the new date
            mov    dl,1           ;yes, set day to 1
            inc    dh             ;and increment month
            cmp    dh,12          ;past end of year?
            jle    month_ok       ;no, set the new date
            mov    dh,1           ;yes, set the month to 1
            inc    cx             ;increment year
month_ok:   set_date cx,dh,dl     ;THIS FUNCTION
```

Get Time (Function 2CH)



Call
AH = 2CH

Return
CH
  Hour (0-23)
CL
  Minutes (0-59)
DH
  Seconds (0 - 59)
DL
  Hundredths (0-99)

Function 2CH returns the current time set in the operating system as binary numbers in CX and DX:

CH Hour (0-23)
CL Minutes (0-59)
DH Seconds (0-59)
DL Hundredths of a second (0-99)

Depending on how your hardware keeps time, some of these fields may be irrelevant. As an example, many CMOS clock chips do not resolve more than seconds. In such a case the value in DL will probably always be 0.

Macro Definition:

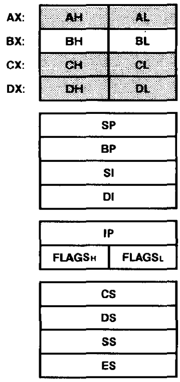
```
get_time  macro
          mov   ah,2CH
          int   21H
          endm
```

Example

The following program continuously displays the time until any key is pressed.

```
time          db    "00:00:00.00",0DH,"$"
;
begin:        get_time                    ;THIS FUNCTION
              byte_to_dec ch,time         ;see end of chapter
              byte_to_dec cl,time[3]      ;see end of chapter
              byte_to_dec dh,time[6]      ;see end of chapter
              byte_to_dec dl,time[9]      ;see end of chapter
              display_time                ;see Function 09H
              check_kbd_status            ;see Function 0BH
              cmp     al,0FFH             ;has a key been pressed?
              je      return              ;yes, terminate
              jmp     begin               ;no, display time
```

Set Time (Function 2DH)



Call
AH = 2DH
CH
  Hour (0-23)
CL
  Minutes (0-59)
DH
  Seconds (0-59)
DL
  Hundredths (0-99)

Return
AL
  00H = Time was valid
  FFH = Time was invalid

Function 2DH sets the time in the operating system. Registers CX and DX must contain a valid time in binary:

- CH Hour (0-23)
- CL Minutes (0-59)
- DH Seconds (0-59)
- DL Hundredths of a second (0-99)

Depending on how your hardware keeps time, some of these fields may be irrelevant. As an example, many CMOS clock chips do not resolve more than seconds. In such a case the value in DL will not be relevant.

If the time is valid, the time is set and AL returns 0. If the time is not valid, the function is canceled and AL returns FFH.

Macro Definition:

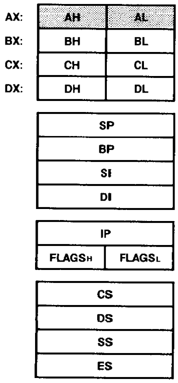
```
set_time  macro  hour,minutes,seconds,hundredths
          mov    ch,hour
          mov    cl,minutes
          mov    dh,seconds
          mov    dl,hundredths
          mov    ah,2DH
          int    21H
          endm
```

Example

The following program sets the system clock to 0 and continuously displays the time. When a character is typed, the display freezes; when another character is typed, the clock is reset to 0 and the display starts again.

```
time            db  "00:00:00.00",0DH,0AH,"$"
;
begin:          set_time  0,0,0,0          ;THIS FUNCTION
read_clock:     get_time                   ;see Function 2CH
                byte_to_dec  ch,time       ;see end of chapter
                byte_to_dec  cl,time[3]    ;see end of chapter
                byte_to_dec  dh,time[6]    ;see end of chapter
                byte_to_dec  dl,time[9]    ;see end of chapter
                display_time               ;see Function 09H
                dir_console_io 0FFH        ;see Function 06H
                cmp       al,00H           ;was a char. typed?
                jne       stop             ;yes, stop the timer
                jmp       read_clock       ;no keep timer on
stop:           read_kbd                   ;see Function 08H
                jmp       begin            ;keep displaying time
```

Set/Reset Verify Flag (Function 2EH)



Call
AH = 2EH
AL
  0 = Do not verify
  1 = Verify

Return
None

Function 2EH tells MS-DOS whether to verify each disk write. If AL is 1, verify is turned on; if AL is 0, verify is turned off. MS-DOS checks this flag each time it writes to a disk.

The flag is normally off; you may wish to turn it on when writing critical data to disk. Because disk errors are rare and verification slows writing, you will probably want to leave it off at other times. You can check the setting with Function 54H (Get Verify State).

Macro Definition:

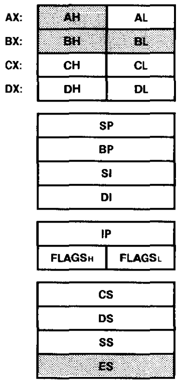
```
verify  macro   switch
        mov     al,switch
        mov     ah,2EH
        int     21H
        endm
```

Example

The following program copies the contents of a single-sided disk in drive A to the disk in drive B, verifying each write. It uses a buffer of 32K bytes.

```
on              equ    1
off             equ    0
;
prompt          db     "Source in A, target in B",0DH,0AH
                db     "Any key to start. $"
first           dw     0
buffer          db     60 dup (512 dup(?))    ;60 sectors
;
begin:          display prompt                ;see Function 09H
                read_kbd                      ;see Function 08H
                verify on                     ;THIS FUNCTION
                mov     cx,6                  ;copy 60 sectors
                                              ;6 times
copy:           push    cx                    ;save counter
                abs_disk_read  0,buffer,60,first ;see Int 25H
                abs_disk_write  1,buffer,64,first ;see Int 26H
                add     first,60              ;do next 60 sectors
                pop     cx                    ;restore counter
                loop    copy                  ;do it again
                verify off                    ;THIS FUNCTION
```

Get Disk Transfer Address (Function 2FH)



Call
AH = 2FH

Return
ES:BX
  Pointer to Disk Transfer Address

Function 2FH returns the segment address of the current Disk Transfer Address in ES and the offset in BX.

Macro Definition: get_dta

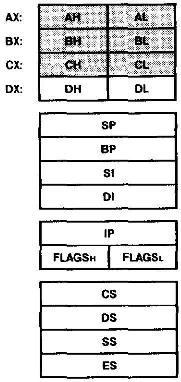
```
get_dta     macro
            mov     ah,2fH
            int     21H
            endm
```

Example

The following program displays the current Disk Transfer Address in the form segment:offset.

```
message   db      "DTA --          :      ",0DH,0AH,"$"
sixteen   db       10H
temp      db       2 dup (?)
;
begin:    get_dta                           ;THIS FUNCTION
          mov     word ptr temp,ex          ;To access each byte
          convert temp[1],sixteen,message[07H]  ;See end of
          convert temp,sixteen,message[09H]    ;chapter for
          convert bh,sixteen,message[0CH]      ;description
          convert bl,sixteen,message[0EH]      ;of CONVERT
          display message                   ;See Function 09H
```

Get MS-DOS Version Number (Function 30H)



Call
AH = 30H

Return
AL
  Major version number
AH
  Minor version number
BH
  OEM serial number
BL:CX
  24-bit user (serial) number

Function 30H returns the MS-DOS version number. AL returns the major version number; AH returns the minor version number. (For example, MS-DOS 3.0 returns 3 in AL and 0 in AH.)

If AL returns 0, the version of MS-DOS is earlier than 2.0.

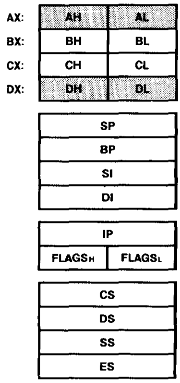
Macro Definition:
```
get_version  macro
             mov    ah,30H
             int    21H
             endm
```

Example

The following program displays the version of MS-DOS if it is 1.28 or greater.

```
message db      "MS-DOS Version  . ",0DH,0AH,"$"
ten     db       0AH                      ;For CONVERT
;
begin:  get_version                       ;THIS FUNCTION
        cmp     al,0                      ;1.28 or later?
        jng     return                    ;No, go home
        convert al,ten,message[0FH]       ;See end of chapter
        convert ah,ten,message[12H]       ;for description
        display message                   ;See Function 9
```

Keep Process (Function 31H)



Call
AH = 31H
AL
  Return code
DX
  Memory size, in paragraphs

Return
None

Function 31H makes a program remain resident after it terminates. It is often used to install device-specific interrupt handlers. Unlike Interrupt 27H (Terminate But Stay Resident), this function request allows more than 64K bytes to remain resident and does not require CS to contain the segment address of the Program Segment Prefix. You should use Function 31H to install a resident program unless it is absolutely imperative that your program be compatible with pre-2.0 versions of MS-DOS.

DX must contain the number of paragraphs of memory required by the program (one paragraph = 16 bytes). AL contains an exit code.

Use of this in .EXE programs requires care. The value in DX must be the total size to remain resident, not just the size of the code segment which is to remain resident. A typical error is to forget about the 100H byte program header prefix and give a value which is 10H in DX which is 10H too small.

MS-DOS terminates the current process and tries to set the memory allocation to the number of paragraphs in DX. No other allocation blocks belonging to the process are released.

The exit code in AL can be retrieved by the parent process with Function 4DH (Get Return Code of Child Process) and can be tested with the IF command using ERRORLEVEL.

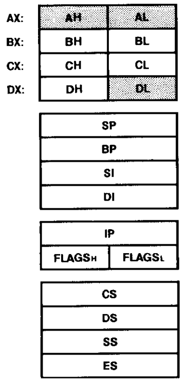
Macro Definition:

```
keep_process  macro return_code,last_byte
              mov   al,return_code
              mov   dx,offset last_byte
              mov   cl,4
              shr   dx,cl
              inc   dx
              mov   ah,31H
              int   21H
              endm
```

Example

Because the most common use of this call is to install a machine-specific routine, an example is not shown. The macro definition shows the calling syntax.

Control-C Check (Function 33H)



Call
AH = 33H
AL
    0 = Get state
    1 = Set state
DL (if AL=1)
    0 = Off
    1 = On

Return
DL (if AL=0)
    0 = Off
    1 = On
AL
    FFH = error (AL was neither 0 nor 1
          when call was made)

Function 33H gets or sets the state of Control-C (or Control-Break for IBM compatibles) checking in MS-DOS. AL must contain a code that specifies the requested action:

0 Return current state of Control-C checking in DL.

1 Set state of Control-C checking to the value in DL.

If AL is 0, DL returns the current state (0=off, 1=on). If AL is 1, the value in DL specifies the state to be set (0=off, 1=on). If AL is neither 0 nor 1, AL returns FFH and the state of Control-C checking is not affected.

MS-DOS normally checks for Control-C only when carrying out certain function requests in the 01H through 0CH group (see the description of specific calls for details). When Control-C checking is on, MS-DOS checks for Control-C when carrying out any function request. For example, if Control-C checking is off, all disk I/O proceeds without interruption; if Control-C checking is on, the Control-C interrupt is issued at the function request that initiates the disk operation.

Note

Programs that use Function Request 06H or 07H to read Control-C as data must ensure that the Control-C checking is off.

Macro Definition:

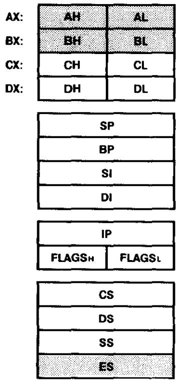
```
ctrl_c_ck  macro   action,state
           mov     al,action
           mov     dl,state
           mov     ah,33H
           int     21H
           endm
```

Example

The following program displays a message that tells whether Control-C checking is on or off:

```
message   db        "Control-C checking ","$"
on        db        "on","$",0DH,0AH,"$"
off       db        "off","$",0DH,0AH,"$"
;
begin:    display   message           ;See Function 09H
          ctrl_c_ck 0                 ;THIS FUNCTION
          cmp       dl,0              ;Is checking off?
          jg        ck_on             ;No
          display   off               ;See Function 09H
          jmp       return            ;Go home
ck_on:    display   on                ;See Function 09H
```

Get Interrupt Vector (Function 35H)



Call
AH = 35H
AL
    Interrupt number

Return
ES:BX
    Pointer to interrupt routine

Function 35H gets the address from the interrupt vector table for the specified interrupt. AL must contain the number of an interrupt.

ES returns the segment address of the interrupt handler; BX returns the offset.

To avoid compatibility problems, programs should never read an interrupt vector directly from memory, nor set an interrupt vector by writing it into memory. Use this function request to get a vector and Function 25H (Set Interrupt Vector) to set a vector, unless it is absolutely imperative that your program be compatible with pre-2.0 versions of MS-DOS.

Macro Definition:

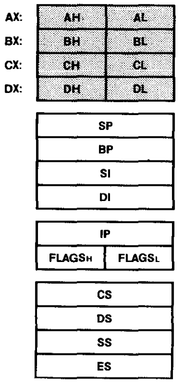
```
get_vector  macro  interrupt
            mov    al,interrupt
            mov    ah,35H
            int    21H
            endm
```

Example

The following program displays the segment and offset (CS:IP) for the handler for Interrupt 25H (Absolute Disk Read).

```
message   db      "Interrupt 25H -- CS:0000 IP:0000"
          db       ODH,OAH,"$"
vec_seg   db       2 dup (?)
vec_off   db       2 dup (?)
;
begin:    push     es                         ;save ES
          get_vector 25H                      ;THIS FUNCTION
          mov      ax,es                      ;INT25H segment in AX
          pop      es                         ;save ES
          convert ax,16,message[20]           ;see end of chapter
          convert bx,16,message[28]           ;see end of chapter
          display message                     ;See Function 9
```

Get Disk Free Space (Function 36H)



Call
AH = 36H
DL
    Drive (0=default, 1=A, etc.)

Return
AX
    0FFFFH if drive number is invalid; otherwise sectors per cluster
BX
    Available clusters
CX
    Bytes per sector
DX
    Clusters per drive

Function 36H returns the number of clusters available on the disk in the specified drive, and sufficient information to calculate the number of bytes available on the disk. DL must contain a drive number (0=default, 1=A, etc.). If the drive number is valid, MD-DOS returns the information in the following registers:

AX	Sectors per cluster
BX	Available clusters
CX	Bytes per sector
DX	Total clusters

If the drive number is invalid, AX returns 0FFFFH.

This call supersedes Functions 1BH and 1CH in earlier versions of MS-DOS.

Macro Definition:

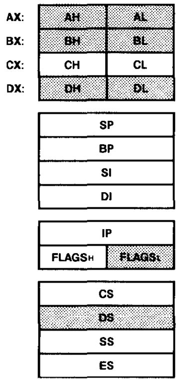
```
get_disk_space  macro   drive
                mov     dl,drive
                mov     ah,36H
                int     21H
                endm
```

Example

The following program displays the space information for the disk in drive B.

```
message   db "       clusters on drive B.",0DH,0AH      ;DX
          db "       clusters available.",0DH,0AH       ;BX
          db "       sectors per cluster.",0DH,0AH      ;AX
          db "       bytes per sector,",0DH,0AH,"$"     ;CX
;
begin:    get_disk_space 2                          ;THIS FUNCTION
          convert    ax,10,message[55]              ;see end of chapter
          convert    bx,10,message[28]              ;see end of chapter
          convert    cx,10,message[83]              ;see end of chapter
          convert    dx,10,message                  ;see end of chapter
          display message                           ;See Function 09H
```

Get Country Data (Function 38H)



Call
AH = 38H
AL
      0 = Current country
      1 to 0FEH = Country code
      0FFH = BX contains Country code
BX (if AL=0FFH)
   Country code 255 or higher
DS:DX
   Pointer to 32-byte memory area

Return
Carry set:
AX
   2 = Invalid country code
Carry not set:
BX
   Country code

Function 38H gets the country-dependent information that MS-DOS uses to control the keyboard and display or sets the currently defined country (to set the country code, see the next function request description). To get the information, DX must contain the offset (from the segment address in DS) of a 32-byte memory area in which the country data is to be returned. AL specifies the country code:

Value in AL	Meaning
0	Retrieve information about the country currently set.
1 to 0FEH	Retrieve information about the country identified by this code.
0FFH	Retrieve information about the country identified by the code in BX.

BX must contain the country code if the code is 255 or greater. The country code is usually the international telephone prefix code.

The country-dependent information is returned in the following form:

Offset Hex	Offset Decimal	Field Name	Length in bytes
00	0	Date format	2 (word)
02	2	Currency symbol	5 (ASCIZ string)
07	7	Thousands separator	2 (ASCIZ string)
09	9	Decimal separator	2 (ASCIZ string)
0B	11	Date separator	2 (ASCIZ string)
0D	13	Time separator	2 (ASCIZ string)
0F	15	Bit field	1
10	16	Currency places	1
11	17	Time format	1
12	18	Case-map call address	4 (dword)
16	22	Data-list separator	2 (ASCIZ string)
18	24	RESERVED	10

Date Format: 0 = USA (m/d/y)
1 = Europe (d/m/y)
2 = Japan (y/m/d)

Bit Field: Bit 0 = 0 Currency symbol precedes amount
1 Currency symbol follows amount

Bit 1 = 0 No space between symbol and amount
1 One space between symbol and amount

All other bits are undefined.

Time format: 0 = 12-hour clock
1 = 24-hour clock

Currency Places: Specifies the number of places that appear after the decimal point on currency amounts.

Case-Mapping Call Address: The segment and offset of a FAR procedure that performs country-specific lowercase-to-uppercase mapping on character values from 80H to 0FFH. You call it with the character to be mapped in AL. If there is an uppercase code for the character, it is returned in AL; if there is not, or if you call it with a value less than 80H in AL, AL is returned unchanged. AL and the FLAGS are the only registers altered.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
2	Invalid country code (no table for it).

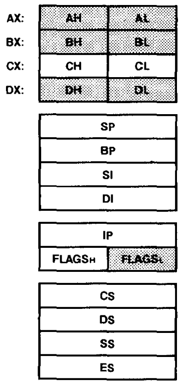
```
Macro Definition: get_country macro  country,buffer
                  local      gc_01
                  mov        dx,offset buffer
                  mov        ax,country
                  cmp        ax,0FFH
                  jl         gc_01
                  mov        al,0FFh
                  mov        bx,country
gc_01:            mov        ah,38h
                  int        21H
                  endm
```

Example

The following program displays the time and date in the format appropriate to the current country code, and the number 999,999 and 99/100 as a currency amount with the proper currency symbol and separators.

```
time       db     "  :  :  ",5 dup (20H),"$"
date       db     "  /  /  ",5 dup (20H),"$"
number     db     "999?999?99",0DH,0AH,"$"
data_area  db      32 dup (?)
;
begin:    get_country  0,data_area        ;THIS FUNCTION
          get_time                        ;See Function 2CH
          byte_to_dec  ch,time            ;See end of chapter
          byte_to_dec  cl,time[03H]       ;for description of
          byte_to_dec  dh,time[06H]       ;CONVERT macro
          get_date                        ;See Function 2AH
          sub        cx,1900              ;Want last 2 digits
          byte_to_dec  cl,date[06H]       ;See end of chapter
          cmp        word ptr data_area,0 ;Check country code
          jne        not_usa              ;It's not USA
          byte_to_dec  dh,date            ;See end of chapter
          byte_to_dec  dl,date[03H]       ;See end of chapter
          jmp        all_done             ;Display data
not_usa:  byte_to_dec  dl,date            ;See end of chapter
          byte_to_dec  dh,date[03H]       ;See end of chapter
all_done: mov        al,data_area[07H]    ;Thousand separator
          mov        number[03H],al       ;Put in NUMBER
          mov        al,data_area[09H]    ;Decimal separator
          mov        number[07H],al       ;Put in AMOUNT
          display    time                 ;See Function 09H
          display    date                 ;See Function 09H
          display_char  data_area[02H]    ;See Function 02H
          display    number               ;See Function 09H
```

Set Country Data (Function 38H)



Call
AH = 38H
DX = -1 (0FFFFH)
AL
  Country code less than 255, or
  0FFH if the country code is in BX
BX (if AL=0FFH)
  Country code 255 or higher

Return
Carry set:
AX
  2 = Invalid country code
Carry not set:
  No error

Function 38H sets the country code that MS-DOS uses to control the keyboard and display, or retrieves the country-dependent information (to get the country data, see the previous function request description). To set the information, DX must contain 0FFFFH. AL must contain the country code if it is less than 255, or 255 to indicate that the country code is in BX. If AL contains 0FFH, BX must contain the country code.

The country code is usually the international telephone prefix code. See the preceding function request description (Get Country Data) for a description of the country data and how it is used.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
2	Invalid country code (no table for it).

Macro Definition:

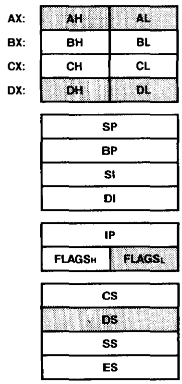
```
set_country macro   country
            local   sc_01
            mov     dx,0FFFFH
            mov     ax,country
            cmp     ax,0FFH
            jl      sc_01
            mov     bx,country
            mov     al,0ffh
sc_01:      mov     ah,38H
            int     21H
            endm
```

Example

The following program sets the country code to the United Kingdom (44).

```
uk          equ             44
;
begin:      set_country uk      ;THIS FUNCTION
            jc              error   ;routine not shown
```

Create Directory (Function 39H)



Call
AH = 39H
DS:DX
    Pointer to pathname

Return
Carry set:
AX
    3 = Path not found
    5 = Access denied
Carry not set:
    No error

Function 39H creates a new subdirectory. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the new subdirectory.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
3	Path not found.
5	No room in the parent directory, a file with the same name exists in the current directory, or the path specifies a device.

Macro Definition:

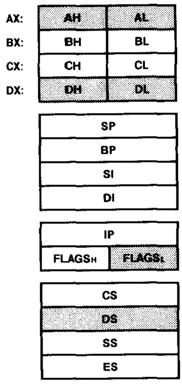
```
make_dir  macro path
          mov   dx,offset path
          mov   ah,39H
          int   21H
          endm
```

Example

The following program adds a subdirectory named NEWDIR to the root directory on the disk in drive B, changes the current directory to NEWDIR, changes the current directory back to the original directory, then deletes NEWDIR. It displays the current directory after each step to confirm the changes.

```
old_path  db          "b:\",0,63 dup (?)
new_path  db          "b:\new_dir",0
buffer    db          "b:\",0,63 dup (?)
;
begin:    get_dir     2,old_path[03H]   ;See Function 47H
          jc          error_get         ;Routine not shown
          display_asciz  old_path       ;See end of chapter
          make_dir    new_path          ;THIS FUNCTION
          jc          error_make        ;Routine not shown
          change_dir new_path           ;See Function 3BH
          jc          error_change      ;Routine not shown
          get_dir     2,buffer[03H]     ;See Function 47H
          jc          error_get         ;Routine not shown
          display_asciz  buffer         ;See end of chapter
          change_dir old_path           ;See Function 3BH
          jc          error_change      ;Routine not shown
          rem_dir     new_path          ;See Function 3AH
          jc          error_rem         ;Routine not shown
          get_dir     2,buffer[03H]     ;See Function 47H
          jc          error_get         ;Routine not shown
          display_asciz  buffer         ;See end of chapter
```

Remove Directory (Function 3AH)



Call
AH = 3AH
DS:DX
    Pointer to pathname

Return
Carry set:
AX
    3 = Path not found
    5 = Access denied
    16 = Current directory
Carry not set:
    No error

Function 3AH deletes a subdirectory. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the subdirectory to be deleted.

The subdirectory must not contain any files. You cannot erase the current directory. If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
3	Path not found.
5	The directory isn't empty; or the path doesn't specify a directory, specifies the root directory, or is invalid.
16	The path specifies the current directory.

Macro Definition:

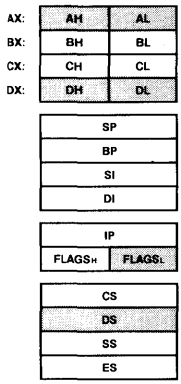
```
rem_dir  macro  path
         mov    dx,offset path
         mov    ah,3AH
         int    21H
         endm
```

Example

The following program adds a subdirectory named NEWDIR to the root directory on the disk in drive B, changes the current directory to NEWDIR, changes the current directory back to the original directory, then deletes NEWDIR. It displays the current directory after each step to confirm the changes.

```
old_path  db         "b:\",0,63 dup (?)
new_path  db         "b:\new_dir",0
buffer    db         "b:\",0,63 dup (?)
;
begin:    get_dir    2,old_path[03H]  ;See Function 47H
          jc         error_get        ;Routine not shown
          display_asciz  old_path    ;See end of chapter
          make_dir   new_path         ;See Function 39H
          jc         error_make       ;Routine not shown
          change_dir new_path         ;See Function 3BH
          jc         error_change     ;Routine not shown
          get_dir    2,buffer[03H]    ;See Function 47H
          jc         error_get        ;Routine not shown
          display_asciz  buffer      ;See end of chapter
          change_dir old_path         ;See Function 3BH
          jc         error_change     ;Routine not shown
          rem_dir    new_path         ;THIS FUNCTION
          jc         error_rem        ;Routine not shown
          get_dir    2,buffer[03H]    ;See Function 47H
          jc         error_get        ;Routine not shown
          display_asciz  buffer      ;See end of chapter
```

Change Current Directory (Function 3BH)



Call
AH = 3BH
DS:DX
  Pointer to pathname

Return
Carry set:
AX
  3 = Path not found
Carry not set:
  No error

Function 3BH changes the current directory. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the new current directory.

The directory string is limited to 64 characters.

If any member of the path doesn't exist, the path is not changed. If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
3	The pathname either doesn't exist or specifies a file, not a directory.

Macro Definition:

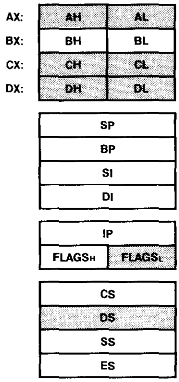
```
change_dir  macro  path
            mov    dx,offset path
            mov    ah,3BH
            int    21H
            endm
```

Example

The following program adds a subdirectory named NEW_DIR to the root directory on the disk in drive B, changes the current directory to NEW_DIR, changes the current directory back to the original directory, then deletes NEW_DIR. It displays the current directory after each step to confirm the changes.

```
old_path   db          "b:\",0,63 dup (?)
new_path   db          "b:\new_dir",0
buffer     db          "b:\",0,63 dup (?)
;
begin:     get_dir     2,old_path[03H]   ;See Function 47H
           jc          error_get         ;Routine not shown
           display_asciz  old_path      ;See end of chapter
           make_dir    new_path          ;See Function 39H
           jc          error_make        ;Routine not shown
           change_dir  new_path          ;THIS FUNCTION
           jc          error_change      ;Routine not shown
           get_dir     2,buffer[03H]     ;See Function 47H
           jc          error_get         ;Routine not shown
           display_asciz  buffer        ;See end of chapter
           change_dir  old_path          ;See Function 3BH
           jc          error_change      ;Routine not shown
           rem_dir     new_path          ;See Function 3AH
           jc          error_rem         ;Routine not shown
           get_dir     2,buffer[03H]     ;See Function 47H
           jc          error_get         ;Routine not shown
           display_asciz  buffer        ;See end of chapter
```

Create Handle (Function 3CH)



Call
AH = 3CH
DS:DX
    Pointer to pathname
CX
    File attribute

Return
Carry set:
AX
    3 = Path not found
    4 = Too many open files
    5 = Access denied
Carry not set:
AX
    Handle

Function 3CH creates a file and assigns it the first available handle. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the file to be created. CX must contain the attribute to be assigned to the file, as described under "File Attributes" earlier in this chapter.

If the specified file does not exist, it is created. If the file does exist, it is truncated to a length of 0. The attribute in CX is assigned to the file and the file is opened for read/write. AX returns the file handle.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
3	The path is invalid.
4	Too many open files (no handle available).
5	Directory full, a directory with the same name exists, or a file with the same name exists with more restrictive attributes.

Macro Definition:

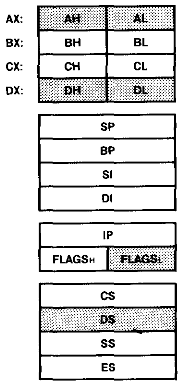
```
create_handle   macro   path,attrib
                mov     dx,offset path
                mov     cx,attrib
                mov     ah,3CH
                int     21H
                endm
```

Example

The following program creates a file named DIR.TMP on the disk in drive B that contains the name and extension of each file in the current directory.

```
srch_file db      "b:*.*",0
tmp_file  db      "b:dir.tmp",0
buffer    db       43 dup (?)
handle    dw       ?
;
begin:     set_dta buffer                      ;See Function 1AH
           find_first_file  srch_file,16H  ;See Function 4EH
           cmp     ax,12H                      ;Directory empty?
           je      all_done                    ;Yes, go home
           create_handle  tmp_file,0           ;THIS FUNCTION
           jc      error                       ;Routine not shown
           mov     handle,ax                   ;Save handle
write_it:  write_handle handle,buffer[1EH],12 ;Function 40H
           find_next_file                      ;See Function 4FH
           cmp     ax,12H                      ;Another entry?
           je      all_done                    ;No, go home
           jmp     write_it                    ;Yes, write record
all_done:  close_handle  handle                ;See Function 3EH
```

Open Handle (Function 3DH)



Call
AH = 3DH
AL
    Access code (see text)
DS:DX
    Pointer to pathname

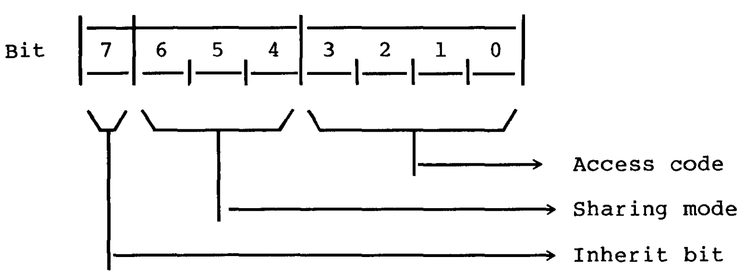
Return
Carry set:
AX
    1 = Invalid function code
    2 = File not found
    3 = Path not found
    4 = Too many open files
    5 = Access denied
    12 = Invalid access
Carry not set:
    No error

Function 3DH opens any file, including hidden and system files, for input or output. DX contains the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the file to be opened. AL contains a code that specifies how the file is to be opened, described later under "Controlling Access to the File."

If there is no error, AX returns the file handle. MS-DOS sets the read/write pointer to the first byte of the file.

Controlling Access to the File

The value in AL is made up of three parts that specify whether the file is to be opened for read, write, or both (access code); what access other processes have to the file (sharing mode); and whether the file is inherited by a child process (inherit bit).



Inherit Bit

The high-order bit (bit 7) specifies whether the file is inherited by a child process created with Function 4BH (Load and Execute Program). If the bit is 0, the file is inherited; if the bit is 1, the file is not inherited.

Sharing Mode

The sharing mode (bits 4-6) specifies what access, if any, other processes have to the open file. It can have the following values:

Bits 4-6	Sharing Mode	Description
000	Compatability	Any process can open the file any number of times with this mode. Fails if the file has been opened with any of the other sharing modes.
001	Deny both	Fails if the file has been opened in compatibility mode or for read or write access, even if by the current process.
010	Deny write	Fails if the file has been opened in compatibility mode or for write access by any other process.
011	Deny read	Fails if the file has been opened in compatibility mode or for read access by any other process.
100	Deny none	Fails if the file has been opened in compatibility mode by any other process.

Access Code

The access code (bits 0-3) specifies how the file is to be used. It can have the following values:

Bits 0-3	Access Allowed	Description
0000	Read	Fails if the file has been opened in deny read or deny both sharing mode.
0002	Write	Fails if the file has been opened in deny write or deny both sharing mode.
0010	Both	Fails if the file has been opened in deny read, deny write, or deny both sharing mode.

If there is an error, the carry flag (CF) is set and the error code is returned in AX.

Code	Meaning
1	File sharing must be loaded to specify a sharing mode (bits 4-6 of AL).
2	The file specified is invalid or doesn't exist.
3	The path specified is invalid or doesn't exist.
4	No handles are available in the current process or the internal system tables are full.
5	The program attempted to open a directory or Volume-ID, or open a read-only file for writing.
12	The access code (bits 0-3 of AL) is not 0, 1, or 2.

If this system call fails because of a file-sharing error, MS-DOS issues Interrupt 24H with error code 2 (Drive Not Ready). A subsequent Function 59H (Get Extended Error) returns the extended error code that specifies a sharing violation.

When opening a file, it is important to inform MS-DOS of any operations other processes may perform on this file (sharing mode). The default (compatibility mode) denies all other processes access to the file. It may be OK for other processes to continue to read the file while your process is operating on it. In this case, you should specify "Deny Write," which inhibits writing by other processes but allows reading them.

Similarly, it is important to specify what operations your process will perform ("Access" mode). The default mode ("Read/write") will cause the open request to fail if another process has the file opened with any sharing mode other than "Deny" mode. If you only want to read the file, your open will succeed unless all other processes have specified "Deny" mode or "Deny write".

Macro Definition:

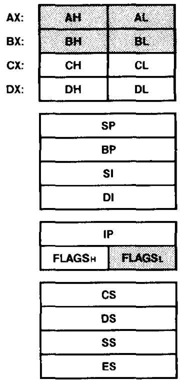
```
open_handle  macro  path,access
             mov    dx, offset path
             mov    al, access
             mov    ah, 3DH
             int    21H
             endm
```

Example

The following program prints the file named TEXTFILE.ASC on the disk in drive B.

```
file        db   "b:textfile.asc",0
buffer      db    ?
handle      dw    ?
;
begin:      open_handle  file,0              ;THIS FUNCTION
            mov  handle,ax                   ;Save handle
read_char:  read_handle handle,buffer,1      ;Read 1 character
            jc   error_read                  ;Routine not shown
            cmp  ax,0                        ;End of file?
            je   return                      ;Yes, go home
            print_char  buffer               ;See Function 05H
            jmp  read_char                   ;Read another
```

Close Handle (Function 3EH)



Call
AH = 3EH
BX
    Handle

Return
Carry set:
AX
    6 = Invalid handle
Carry not set:
    No error

Function 3EH closes a file opened with Function 3DH (Open Handle) or 3CH (Create Handle). BX must contain the handle of the open file that is to be closed.

If there is no error, MS-DOS closes the file and flushes all internal buffers. If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
6	Handle is not open or is invalid.

Macro Definition:

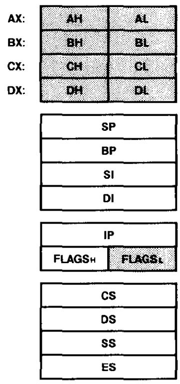
```
close_handle  macro  handle
              mov    bx,handle
              mov    ah,3EH
              int    21H
              endm
```

Example

The following program creates a file named DIR.TMP in the current directory on the disk in drive B that contains the filename and extension of each file in the current directory.

```
srch_file  db   "b:*.*",0
tmp_file   db   "b:dir.tmp",0
buffer     db    43 dup (?)
handle     dw    ?
;
begin:     set_dta  buffer                   ;See Function 1AH
           find_first_file  srch_file,16H  ;See Function 4EH
           cmp      ax,12H                   ;Directory empty?
           je       all_done                 ;Yes, go home
           create_handle  tmp_file,0         ;See Function 3CH
           jc       error_create             ;Routine not shown
           mov      handle,ax                ;Save handle
write_it:  write_handle  handle,buffer[1EH],12 ;See Function
           jc       error_write                        ;40H
           find_next_file                    ;See Function 4FH
           cmp      ax,12H                   ;Another entry?
           je       all_done                 ;No, go home
           jmp      write_it                 ;Yes, write record
all_done:  close_handle  handle              ;See Function 3EH
           jc       error_close              ;Routine not shown
```

Read Handle (Function 3FH)



Call
AH = 3FH
BX
    Handle
CX
    Bytes to read
DS:DX
    Pointer to buffer

Return
Carry set:
AX
    5 = Access denied
    6 = Invalid handle
Carry not set:
AX
    Bytes read

Function 3FH reads from the file or device associated with the specified handle. BX must contain the handle. CX must contain the number of bytes to be read. DX must contain the offset (to the segment address in DS) of the buffer.

If there is no error, AX returns the number of bytes read; if you attempt to read starting at end of file, AX returns 0. The number of bytes specified in CX is not necessarily transferred to the buffer; if you use this call to read from the keyboard, for example, it reads only up to the first CR.

If you use this function request to read from standard input, the input can be redirected.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
5	Handle is not open for reading.
6	Handle is not open or is invalid.

Macro Definition:

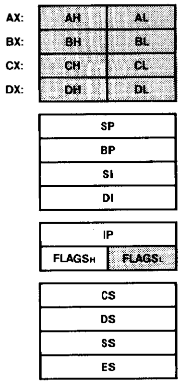
```
read_handle  macro   handle,buffer,bytes
             mov     bx,handle
             mov     dx,offset buffer
             mov     cx,bytes
             mov     ah,3FH
             int     21H
             endm
```

Example

The following program displays the file named TEXTFILE.ASC on the disk in drive B.

```
filename    db      "b:\textfile.asc",0
buffer      db       129 dup (?)
handle      dw       ?
;
begin:      open_handle  filename,0           ;See Function 3DH
            jc           error_open           ;Routine not shown
            mov          handle,ax            ;Save handle
read_file:  read_handle  buffer,file_handle,128
            jc           error_open           ;Routine not shown
            cmp          ax,0                 ;End of file?
            je           return               ;Yes, go home
            mov          bx,ax                ;# of bytes read
            mov          buffer[bx],"$"       ;Make a string
            display      buffer               ;See Function 09H
            jmp          read_file            ;Read more
```

Write Handle (Function 40H)



Call
AH = 40H
BX
   Handle
CX
   Bytes to write
DS:DX
   Pointer to buffer

Return
Carry set:
AX
   5 = Access denied
   6 = Invalid handle
Carry not set:
AX
   Bytes written

Function 40H writes to the file or device associated with the specified handle. BX must contain the handle. CX must contain the number of bytes to be written. DX must contain the offset (to the segment address in DS) of the data to be written.

If there is no error, AX returns the number of bytes written. Be sure to check AX after writing to a disk file: if it contains 0, the disk is full; if its value is less than the number in CX when the call was made, it indicates an error even though the carry flag isn't set.

If you use this function request to write to standard output, the output can be redirected. If you call this function request with CX=0, the file size is set to the value of the read/write pointer. Allocation units are allocated or released, as required, to satisfy the new file size.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
5	Handle is not open for writing.
6	Handle is not open or is invalid.

Macro Definition: write_handle

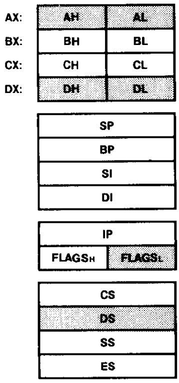
```
write_handle  macro   handle,data,bytes
              mov     bx,handle
              mov     dx,offset data
              mov     cx,bytes
              mov     ah,40H
              int     21H
              endm
```

Example

The following program creates a file named DIR.TMP in the current directory on the disk in drive B that contains the filename and extension of each file in the current directory.

```
srch_file db     "b:*.*",0
tmp_file  db     "b:dir.tmp",0
buffer    db      43 dup (?)
handle    dw      ?
;
begin:    set_dta buffer                       ;See Function 1AH
          find_first_file  srch_file,16H ;Check directory
          cmp     ax,12H                       ;Directory empty?
          je      return                       ;Yes, go home
          create_handle  tmp_file,0            ;See Function 3CH
          jc      error_create                 ;Routine not shown
          mov     handle,ax                    ;Save handle
write_it: write_handle handle,buffer[1EH],12 ;THIS FUNCTION
          jc      error_write                  ;Routine not shown
          find_next_file                       ;Check directory
          cmp     ax,12H                       ;Another entry?
          je      all_done                     ;No, go home
          jmp     write_it                     ;Yes, write record
all_done: close_handle  handle                 ;See Function 3EH
          jc      error_close                  ;Routine not shown
```

Delete Directory Entry (Function 41H)



Call
AH = 41H
DS:DX
    Pointer to pathname

Return
Carry set:
AX
    2 = File not found
    5 = Access denied
Carry not set:
    No error

Function 41H erases a file by deleting its directory entry. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the file to be deleted. Wildcard characters cannot be used.

If the file exists and is not read-only, it is deleted. If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
2	Path is invalid or file doesn't exist.
5	Path specifies a directory or read-only file.

To delete a file with the read-only attribute, first change its attribute to 0 with Function 43H (Get/Set File Attribute).

Macro Definition:

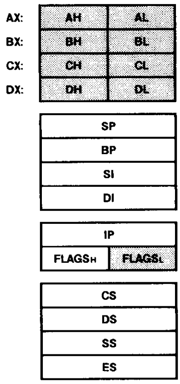
```
delete_entry  macro  path
              mov    dx,offset path
              mov    ah,41H
              int    21H
              endm
```

Example

The following program deletes all files on the disk in drive B whose date is earlier than December 31, 1981.

```
year        db        1981
month       db        12
day         db        31
files       db        ?
ten         db        0AH
message     db       "NO FILES DELETED.",0DH,0AH,"$"
path        db       "b:*.*", 0
buffer      db        43 dup (?)
;
begin:      set_dta  buffer              ;See Function 1AH
            select_disk "B"              ;See Function 0EH
            find_first_file  path,0      ;See Function 4EH
            jc        all_done           ;Go home if empty
compare:    convert_date  buffer         ;See end of chapter
            cmp       cx,year            ;After 1981?
            jg        next               ;Yes, don't delete
            cmp       dl,month           ;After December?
            jg        next               ;Yes, don't delete
            cmp       dh,day             ;31st or after?
            jge       next               ;Yes, don't delete
            delete_entry  buffer[1EH]  ;THIS FUNCTION
            jc        error_delete       ;Routine not shown
            inc       files              ;Bump file counter
next:       find_next_file               ;Check directory
            jnc       compare            ;Go home if done
how_many:   cmp       files,0            ;Was directory empty?
            je        all_done           ;Yes, go home
            convert  files,ten,message ;See end of chapter
all_done:   display  message             ;See Function 09H
            select_disk "A"              ;See Function 0EH
```

Move File Pointer (Function 42H)



Call
AH = 42H
AL
    Method of moving
BX
    Handle
CX:DX
    Distance in bytes (offset)

Return
Carry set:
AX
    1 = Invalid function
    6 = Invalid handle
Carry not set:
DX:AX
    New read/write pointer location

Function 42H moves the read/write pointer of the file associated with the specified handle. BX must contain the handle. CX and DX must contain a 32-bit offset (CX contains the most significant byte). AL must contain a code that specifies how to move the pointer:

Code	Cursor Is Moved To
0	Beginning of file plus the offset.
1	Current pointer location plus the offset.
2	End of file plus the offset.

DX and AX return the new location of the read/write pointer (a 32-bit integer; DX contains the most significant byte). You can determine the length of a file by setting CX:DX to 0, AL to 2, and calling this function request; DX:AX return the offset of the byte after the last byte in the file (size of the file in bytes).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL isn't 0, 1, or 2.
6	Handle isn't open.

Macro Definition:

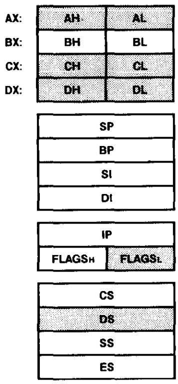
```
move_ptr  macro  handle,high,low,method
          mov    bx,handle
          mov    cx,high
          mov    dx,low
          mov    al,method
          mov    ah,42H
          int    21H
          endm
```

Example

The following program prompts for a letter, converts the letter to its alphabetic sequence (A=1, B=2, etc.), then reads and displays the corresponding record from the file named ALPHABET.DAT in the current directory on the disk in drive B. The file contains 26 records; each record is 28 bytes long.

```
file          db       "b:alphabet.dat",0
buffer        db        28 dup (?),"$"
prompt        db       "Enter letter: $"
crlf          db        0DH,0AH,"$"
handle        db        ?
record_length  dw   28
;
begin:    open_handle  file,0      ;See Function 3DH
          jc       error_open      ;Routine not shown
          mov      handle,ax       ;Save handle
get_char: display  prompt          ;See Function 09H
          read_kbd_and_echo        ;See Function 01H
          sub      al,41h          ;Convert to sequence
          mul      byte ptr record_length  ;Calculate offset
          move_ptr handle,0,ax,0   ;THIS FUNCTION
          jc       error_move      ;Routine not shown
          read_handle handle,buffer,record_length
          jc       error_read      ;Routine not shown
          cmp      ax,0            ;End of file?
          je       return          ;Yes, go home
          display  crlf            ;See Function 09H
          display  buffer          ;See Function 09H
          display  crlf            ;See Function 09H
          jmp      get_char        ;Get another character
```

Get/Set File Attributes (Function 43H)



Call
AH = 43H
AL
    0 = Get attributes
    1 = Set attributes
CX (if AL=1)
    Attributes to be set
DS:DX
    Pointer to pathname

Return
Carry set:
AX
    1 = Invalid function
    3 = Path not found
    5 = Access denied
Carry not set:
CX
    Attribute byte (if AL=0)

Function 43H gets or sets the attributes of a file. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of a file. AL must specify whether to get or set the attribute (0=get, 1=set).

If AL is 0 (get the attribute), the attribute byte is returned in CX. If AL is 1 (set the attribute), CX must contain the attributes to be set. The attributes are described under "File Attributes" earlier in this chapter.

You cannot change the volume-ID bit (08H) or the directory bit (10H) of the attribute byte with this function request.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL isn't 0 or 1.
3	Path is invalid or file doesn't exist.
5	Attribute in CX cannot be changed (directory or Volume-ID).

Macro Definition:

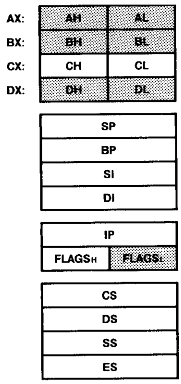
```
change_attr  macro  path,action,attrib
             mov    dx,offset path
             mov    al,action
             mov    cx,attrib
             mov    ah,43H
             int    21H
             endm
```

Example

The following program displays the attributes assigned to the file named REPORT.ASM in the current directory on the disk in drive B.

```
header    db      15 dup (20h),"Read-",0DH,0AH
          db     "Filename       Only       Hidden     "
          db     "System    Volume    Sub-Dir    Archive"
          db      0DH,0AH,0DH,0AH,"$"
path      db     "b:report.asm",3 dup (0),"$"
attribute dw      ?
blanks    db      9 dup (20h),"$"
;
begin:    change_attr  path,0,0  ;THIS FUNCTION
          jc      error_mode     ;Routine not shown
          mov     attribute,cx   ;Save attribute byte
          display header         ;See Function 09H
          display path           ;See Function 09H
          mov     cx,6           ;Check 6 bits (0-5)
          mov     bx,1           ;Start with bit 0
chk_bit:  test    attribute,bx   ;Is the bit set?
          jz      no_attr        ;No
          display_char "X"       ;See Function 02H
          jmp short  next_bit    ;Done with this bit
no_attr:  display_char  20h      ;See Function 02H
next_bit: display_blanks         ;See Function 09H
          shl     bx,1           ;Move to next bit
          loop    chk_bit        ;Check it
```

IOCTL Data (Function 44H, Codes 0 and 1)



Call
AH = 44H
AL
  0 = Get device data
  1 = Set device data
BX
  Handle
DX
  Device data (see text)

Return
Carry set:
AX
  1 = Invalid function
  6 = Invalid handle
Carry not set:
DX
  Device data

Function 44H, Codes 0 and 1 either gets or sets the data MS-DOS uses to control the device. AL must contain 0 to get the data or 1 to set it. BX must contain the handle. If AL is 1, DH must contain 0.

The device data word is specified or returned in DX. If bit 7 of the data is 1, the handle refers to a device and the other bits have the following meanings:

Bit	Value	Meaning
15		RESERVED.
14	1	Device can process control strings sent with Function 44H, Codes 2 and 3 (IOCTL Control). This bit can only be read; it cannot be set.
13-8		RESERVED
6	0	End of file on input.
5	1	Don't check for control characters.
	0	Check for control characters.
4	1	RESERVED.
3	1	Clock device.
2	1	Null device.
1	1	Console output device.
0	1	Console input device.

The control characters referred to in the description of bit 5 are Control-C, Control-P, Control-S, and Control-Z. To read these characters as data, rather than having them interpreted as control characters, bit 5 must be set and

Control-C checking must be turned off, either with Function 33H (Control-C Check) or the MS-DOS Break command.

If bit 7 of DX is 0, the handle refers to a file and the other bits have the following meanings:

Bit	Value	Meaning
15-8		RESERVED
6	0	The file has been written.
0-5		Drive number (0=A, 1=B, etc.).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL is not 0 or 1, or AL is 1 but DH is not 0.
6	The handle in BX is not open or invalid.

Macro Definition:

```
ioctl_data macro  code,handle
           mov    bx,handle
           mov    al,code
           mov    ah,44H
           int    21H
           endm
```

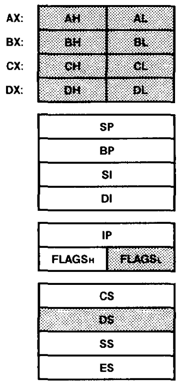
Example

The following program gets the device data for Standard Output and sets the bit that specifies not to check for control characters (bit 5), then clears the bit.

```
get     equu        0
```

```
get     equ         0
set     equ         1
stdout  equ         1
;
begin:  ioctl_data  get,stdout          ;THIS FUNCTION
        jc          error               ;routine not shown
        mov         dh,0                ;clear DH
        or          dl,20H              ;set bit 5
        ioctl_data  set,stdout          ;THIS FUNCTION
        jc          error               ;routine not shown
;
;  <control characters now treated as data, or "raw mode">
;
        ioctl_data  get,stdout          ;THIS FUNCTION
        jc          error               ;routine not shown
        mov         dh,0                ;clear DH
        and         dl,0DFH             ;clear bit 5
        ioctl_data  set,stdout          ;THIS FUNCTION
;
; <control characters now interpreted, or "cooked mode">
;
```

IOCTL Character (Function 44H, Codes 2 and 3)



Call
AH = 44H
AL
   2 = Send control data
   3 = Receive control data
BX
   Handle
CX
   Bytes to read or write
DS:DX
   Pointer to buffer

Return
Carry set:
AX
   1 = Invalid function
   6 = Invalid handle
Carry not set:
AX
   Bytes transferred

Function 44H, Codes 2 and 3 send or receive control data to or from a character device. AL must contain 2 to send data or 3 to receive. BX must contain the handle of a character device, such as a printer or serial port. CX must contain the number of bytes to be read or written. DX must contain the offset (to the segment address in DS) of the data buffer.

AX returns the number of bytes transferred. The device driver must be written to support the IOCTL interface.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL is not 2 or 3, or the device cannot perform the specified function.
6	The handle in BX isn't open or doesn't exist.

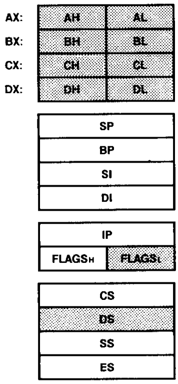
Macro Definition:

```
ioctl_char  macro   code,handle,buffer
            mov     bx,handle
            mov     dx,offset buffer
            mov     al,code
            mov     ah,44H
            int     21H
            endm
```

Example

Because processing of IOCTL control data depends on the device and device driver, no example is included.

IOCTL Block (Function 44H, Codes 4 and 5)



Call
AH = 44H
AL
    4 = Send control data
    5 = Receive control data
BL
    Drive number (0=default, 1=A, etc.)
CX
    Bytes to read or write
DS:DX
    Pointer to buffer

Return
Carry set:
AX
    1 = Invalid function
    5 = Invalid drive
Carry not set:
AX
    Bytes transferred

Function 44H, Codes 4 and 5 send or receive control data to or from a block device. AL must contain 4 to send data or 5 to receive. BL must contain the drive number (0=default, 1=A, etc.). CX must contain the number of bytes to be read or written. DX must contain the offset (to the segment address in DS) of the data buffer.

AX returns the number of bytes transferred. The device driver must be written to support the IOCTL interface. To determine this, use Function 44H, Code 0 to get the device data and test bit 14; if it is set, the driver supports IOCTL.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL is not 4 or 5, or the device cannot perform the specified function.
5	The number in BL is not a valid drive number.

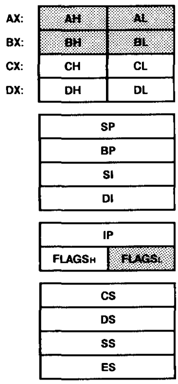
Macro Definition:

```
ioctl_block  macro  code,drive,buffer
             mov    bl,drive
             mov    dx,offset buffer
             mov    al,code
             mov    ah,44H
             int    21H
             endm
```

Example

Because processing of IOCTL control data depends on the device and device driver, no example is included.

IOCTL Status (Function 44H, Codes 6 and 7)



Call
AH = 44H
AL
  6 = Check input status
  7 = Check output status
BX
  Handle

Return
Carry set:
AX
  1 = Invalid function
  5 = Access denied
  6 = Invalid handle
  13 = Invalid data
Carry not set:
AL
  00H = Not ready
  0FFH = Ready

Function 44H, Codes 6 and 7 check whether the file or device associated with a handle is ready. AL must contain 6 to check whether the handle is ready for input or 7 to check whether the handle is ready for output. BX must contain the handle.

AL returns the status:

Value	Meaning for Device	Meaning for Input File	Meaning for Output File
00H	Not ready	Pointer is at EOF	Ready
0FFH	Ready	Ready	Ready

An output file always returns ready, even if the disk is full.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL is not 6 or 7.
5	Access denied.
6	The number in BX isn't a valid, open handle.
13	Invalid data.

Macro Definition:

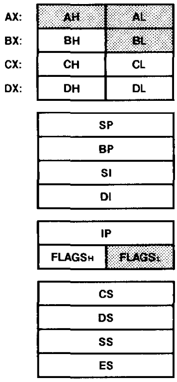
```
ioctl_status    macro   code,handle
                mov     bx,handle
                mov     al,code
                mov     ah,44H
                int     21H
                endm
```

Example

The following program displays a message that tells whether the file associated with handle 6 is ready for input or at end-of-file.

```
stdout      equu          1
;
message     db            "File is "
ready       db            "ready."
at_eof      db            "at EOF."
crlf        db            ODH,OAH
;
begin:      write_handle stdout,message,8    ;display message
            jc           write_error         ;routine not shown
            ioctl_status 6                   ;THIS FUNCTION
            jc           ioctl_error         ;routine not shown
            cmp          al,0                ;check status code
            jne          not_eof             ;file is ready
            write_handle stdout,at_eof,7     ;see Function 40H
            jc           write_error         ;routine not shown
            jmp          all_done            ;clean up & go home
not_eof:    write_handle stdout,ready,6      ;see Function 40H
all_done:   write_handle stdout,crlf,2       ;see Function 40H
            jc           write_error         ;routine not shown
```

IOCTL Is Changeable (Function 44H, Code 08H)



Call
AH = 44H
AL = 08H
BL
   Drive number (0=default, 1=A, etc.)

Return
Carry set:
AX
   1 = Invalid function
   15 = Invalid drive
Carry not set:
AX
   0 = Changeable
   1 = Not changeable

Function 44H, Code 08H checks whether a drive contains a fixed or removable disk. BL must contain the drive number (0=default, 1=A, etc.). AX returns 0 if the disk can be changed, 1 if it cannot.

This call lets a program determine whether to issue a message to change disks.

If there is an error, the carry flag (CF) is set and the error code is returned in AX.

Code	Meaning
1	The device does not support this call.
15	The number in BL is not a valid drive number.

In the case where this call returns error 1 because the device doesn't support the call, the caller should make the assumption that the driver cannot be changed.

Macro Definition:

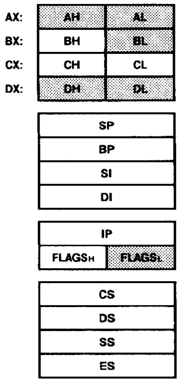
```
ioctl_change  macro   drive
              mov     bl, drive
              mov     al, 08H
              mov     ah, 44H
              int     21H
              endm
```

Example

The following program checks whether the current drive contains a removable disk. If not, processing continues; if so, it prompts the user to replace the disk in the current drive.

```
stdout      equ         1
;
message     db          "Please replace disk in drive "
drives      db          "ABCD"
crlf        db           0DH,0AH
;
begin:      ioctl_change 0            ;THIS FUNCTION
            jc          ioctl_error ;routine not shown
            cmp         ax,0          ;current drive changeable?
            jne         continue      ;no, continue processing
            write_handle stdout,message,29 ;see Function 40H
            jc          write_error ;routine not shown
            current_disk              ;see Function 19H
            xor         bx,bx         ;clear index
            mov         bl,al         ;get current drive
            display_char drives[bx] ;see Function 02H
            write_handle stdout,crlf,2 ;see Function 40H
            jc          write_error ;routine not shown
continue:
;           (Further processing here)
```

IOCTL Is Redirected Block (Function 44H, Code 09H)



Call
AH = 44H
AL = 09H
BL
    Drive number (0=default, 1=A, etc.)

Return
Carry set:
AX
    1 = Invalid function code
    15 = Invalid drive number
Carry not set:
DX
    Device attribute bits

Function 44H, Code 09H checks whether a drive letter refers to a drive on a Microsoft Networks workstation (local) or is redirected to a server (remote). BL must contain the drive number (0=default, 1=A, etc.).

If the block device is local, DX returns the attribute word from the device header. If the block device is remote, only bit 12 (1000h) is set; the other bits are 0 (reserved).

An application program should not test bit 12. Applications should make no distinction between local and remote files or devices.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	File sharing must be loaded to use this system call.
15	The number in BL is not a valid drive number.

Macro Definition:

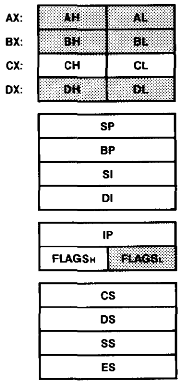
```
ioctl_rblock macro  drive
             mov    bl, drive
             mov    al, 09H
             mov    ah, 44H
             int    21H
             endm
```

Example

The following program checks whether drive B is local or remote, and displays the appropriate message.

```
stdout      equu        1
;
message     db          "Drive B: is "
loc         db          "local."
rem         db          "remote."
crlf        db           0DH,0AH
;
begin:      write_handle stdout,message,12  ;display message
            jc           write_error        ;routine not shown
            ioctl_rblock 2                  ;THIS FUNCTION
            jc           ioctl_error        ;routine not shown
            test         dx,1000h           ;bit 12 set?
            jnz          not_loc            ;yes, it's remote
            write_handle stdout,loc,6       ;see Function 40H
            jc           write_error        ;routine not shown
            jmp          done
not_loc:    write_handle stdout,rem,7       ;see Function 40H
            jc           write_error        ;routine not shown
done:       write_handle stdout,crlf,2      ;see Function 40H
            jc           write_error        ;routine not shown
```

IOCTL Is Redirected Handle (Function 44H, Code 0AH)



Call
AH = 44H
AL = 0AH
BX
  Handle

Return
Carry set:
AX
  1 = Invalid function code
  6 = Invalid handle
Carry not set:
DX
  IOCTL bit field

Function 44H, Code 0AH checks whether a handle refers to a file or device on a Microsoft Networks workstation (local) or is redirected to a server (remote). BX must contain the file handle. DX returns the IOCTL bit field; Bit 15 is set if the handle refers to a remote file or device.

An application program should not test bit 15. Applications should make no distinction among local and remote files and devices.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	Network must be loaded to use this system call.
6	The handle in BX is not a valid, open handle.

Macro Definition:

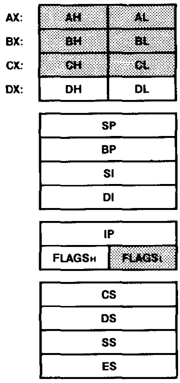
```
ioctl_rhandle macro  handle
              mov    bx, handle
              mov    al, 0AH
              mov    ah, 44H
              int    21H
              endm
```

Example

The following program checks whether handle 5 refers to a local or remote file or device, then displays the appropriate message.

```
stdout     equ        1
;
message    db         "Handle 5 is "
loc        db         "local."
rem        db         "remote."
crlf       db          0DH,0AH
;
begin:     write_handle stdout,message,12;display message
           jc           write_error      ;routine not shown
           ioctl_rhandle 5               ;THIS FUNCTION
           jc           ioctl_error      ;routine not shown
           test         dx,1000h         ;bit 12 set?
           jnz          not_loc          ;yes, it's remote
           write_handle stdout,loc,6     ;see Function 40H
           jc           write_error      ;routine not shown
           jmp          done
not_loc:   write_handle stdout,rem,7     ;see Function 40H
           jc           write_error      ;routine not shown
done:      write_handle stdout,crlf,2    ;see Function 40H
           jc           write_error      ;routine not shown
```

IOCTL Retry (Function 44H, Code 0BH)



Call
AH = 44H
AL = 0BH
BX
    Number of retries
CX
    Wait time

Return
Carry set:
AX
    1 = Invalid function code
Carry not set:
    No error

Function 44H, Code 0BH specifies how many times MS-DOS should retry a disk operation that fails because of a file-sharing violation. BX must contain the number of retries. CX controls the pause between retries.

MS-DOS retries a disk operation that fails because of a file-sharing violation three times unless this system call is used to specify a different number. After the specified number of retries, MS-DOS issues Interrupt 24 for the requesting process.

The effect of the delay parameter in CX is machine-dependent because it specifies how many times MS-DOS should execute an empty loop. The actual time varies, depending on the processor and clock speed. You can determine the effect on your machine by using Debug to set the retries to 1 and time several values of CX.

If there is an error, the carry flag (CF) is set and the error code is returned in AX.

Code	Meaning
1	File sharing must be loaded to use this system call.

Macro Definition:

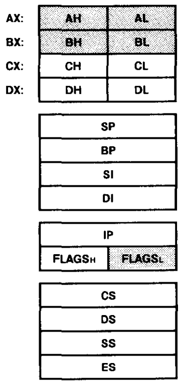
```
ioctl_retry macro  retries, wait
            mov    bx, retries
            mov    cx, wait
            mov    al, 0BH
            mov    ah, 44H
            int    21H
            endm
```

Example

The following program sets the number of sharing retries to 10 and specifies a delay of 1000 between retries.

```
begin:    ioctl_retry  10,1000          ;THIS FUNCTION
          jc           error            ;routine not shown
```

Duplicate File Handle (Function 45H)



Call
AH = 45H
BX
    Handle

Return
Carry set:
AX
    4 = Too many open files
    6 = Invalid handle
Carry not set:
AX
    New handle

Function 45H creates an additional handle for a file. BX must contain the handle of an open file.

MS-DOS returns the new handle in AX. The new handle refers to the same file as the handle in BX, with the file pointer at the same position.

After this function request, moving the read/write pointer of either handle also moves the pointer for the other handle. This function request is usually used to redirect standard input (handle 0) and standard output (handle 1). For a description of standard input, standard output, and the advantages and techniques of manipulating them, see Software Tools by Brian W. Kernighan and P.J. Plauger (Addison-Wesley Publishing Co., 1976).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
4	Too many open files (no handle available).
6	Handle is not open or is invalid.

Macro Definition:

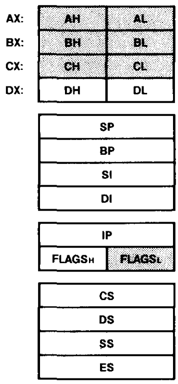
```
xdup    macro   handle
        mov     bx,handle
        mov     ah,45H
        int     21H
        endm
```

Example

The following program redefines standard output (handle 1) to a file named DIRFILE, invokes a second copy of COMMAND.COM to list the directory (which writes the directory to DIRFILE), then restores standard input to handle 1.

```
pgm_file  db    "command.com",0
cmd_line  db    9,"/c dir /w",0dH
parm_blk  db    14 dup (0)
path            db  "dirfile",0
dir_file        dw      ?        ; For handle
sav_stdout dw   ?                ; For handle
;
begin:    set_block  last_inst   ; See Function 4AH
          jc      error_setblk   ; Routine not shown
          create_handle  path,0  ; See Function 3CH
          jc      error_create   ; Routine not shown
          mov     dir_file,ax    ; Save handle
          xdup    1              ; THIS FUNCTION
          jc      error_xdup     ; Routine not shown
          mov     sav_stdout,ax  ; Save handle
          xdup2   dir_file,1     ; See Function 46H
          jc      error_xdup2    ; Routine not shown
          exec    pgm_file,cmd_line,parm_blk ; See Function
                                                   4BH
          jc      error_exec     ; Routine not shown
          xdup2   sav_stdout,1   ; See Function 46H
          jc      error_xdup2    ; Routine not shown
          close_handle sav_stdout ; See Function 3EH
          jc      error_close    ; Routine not shown
          close_handle dir_file  ; See Function 3EH
          jc      error_close    ; Routine not shown
```

Force Duplicate File Handle (Function 46H)



Call
AH = 46H
BX
    Handle
CX
    Second handle

Return
Carry set:
AX
    4 = Too many open files
    6 = Invalid handle
Carry not set:
    No error

Function 46H forces a specified handle to refer the same file as another handle already associated with an open file. BX must contain the handle of the open file; CX must contain the second handle.

On return, the handle in CX now refers to the same file at the same position as the handle in BX. If the file referred to by the handle in CX was open at the time of the call, it is closed.

After this call, moving the read/write pointer of either handle also moves the pointer for the other handle. This function request is normally used to redirect standard input (handle 0) and standard output (handle 1). For a description of standard input, standard output, and the advantages and techniques of manipulating them, see Software Tools by Brian W. Kernighan and P.J. Plauger (Addison-Wesley Publishing Co., 1976).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
4	Too many open files (no handle available).
6	Handle is not open or is invalid.

Macro Definition:

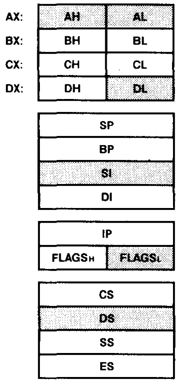
```
xdup2   macro   handle1,handle2
        mov     bx,handle1
        mov     cx,handle2
        mov     ah,46H
        int     21H
        endm
```

Example

The following program redefines standard output (handle 1) to a file named DIRFILE, invokes a second copy of COMMAND.COM to list the directory (which writes the directory to DIRFILE), then restores standard input to handle 1.

```
pgm_file  db    "command.com",0
cmd_line  db    9,"/c dir /w",0dH
parm_blk  db    14 dup (0)
path            db  "dirfile",0
dir_file        dw      ?           ; For handle
sav_stdout dw   ?                   ; For handle
;
begin:    set_block  last_inst  ; See Function 4AH
          jc      error_setblk  ; Routine not shown
          create_handle  path,0 ; See Function 3CH
          jc      error_create  ; Routine not shown
          mov     dir_file,ax   ; Save handle
          xdup    1             ; See Function 45H
          jc      error_xdup    ; Routine not shown
          mov     sav_stdout,ax ; Save handle
          xdup2   dir_file,1    ;
          jc      error_xdup2   ; Routine not shown
          exec    pgm_file,cmd_line,parm_blk ; See Function
                                                     4BH
          jc      error_exec    ; Routine not shown
          xdup2   sav_stdout,1  ; THIS FUNCTION
          jc      error_xdup2   ; Routine not shown
          close_handle sav_stdout ; See Function 3EH
          jc      error_close   ; Routine not shown
          close_handle dir_file ; See Function 3EH
          jc      error_close   ; Routine not shown
```

Get Current Directory (Function 47H)



Call
AH = 47H
DS:SI
    Pointer to 64-byte memory area
DL
    Drive number

Return
Carry set:
AX
    15 = Invalid drive number
Carry not set:
    No error

Function 47H returns the pathname of the current directory on a specified drive. DL must contain a drive number (0=default, 1=A, etc.). SI must contain the offset (from the segment address in DS) of a 64-byte memory area.

MS-DOS places an ASCIZ string in the memory area that consists of the pathname, starting from the root directory, of the current directory for the drive specified in DL. The string does not begin with a backslash and does not include the drive letter.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
15	The number in DL is not a valid drive number.

Macro Definition:

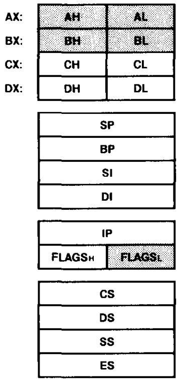
```
get_dir  macro   drive,buffer
         mov     dl,drive
         mov     si,offset buffer
         mov     ah,47H
         int     21H
         endm
```

Example

The following program displays the current directory on the disk in drive B.

```
disk          db       "b:\$"
buffer        db        64 dup (?)
;
begin:        get_dir  2,buffer           ;THIS FUNCTION
              jc       error_dir          ;Routine not shown
              display  disk               ;See Function 09H
              display_asciz  buffer      ;See end of chapter
```

Allocate Memory (Function 48H)



Call
AH = 48H
BX
  Paragraphs of memory requested

Return
Carry set:
AX
  7 = Memory control blocks damaged
  8 = Insufficient memory
BX
  Paragraphs of memory available
Carry not set:
AX
  Segment address of allocated memory

Function 48H tries to allocate the specified amount of memory to the current process. BX must contain the number of paragraphs of memory (1 paragraph is 16 bytes).

If sufficient memory is available to satisfy the request, AX returns the segment address of the allocated memory (the offset is 0). If sufficient memory is not available, BX returns the number of paragraphs of memory in the largest available block.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
7	Memory control blocks damaged (a user program changed memory that doesn't belong to it).
8	Not enough free memory to satisfy the request.

Macro Definition:

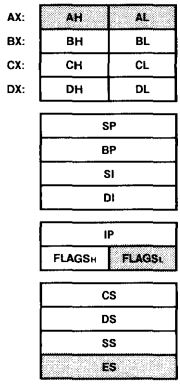
```
allocate_memory  macro   bytes
                 mov     bx,bytes
                 mov     cl,4
                 shr     bx,cl
                 inc     bx
                 mov     ah,48H
                 int     21H
                 endm
```

Example

The following program opens the file named TEXTFILE.ASC, calculates its size with Function 42H (Move File Pointer), allocates a block of memory the size of the file, reads the file into the allocated memory block, then frees the allocated memory.

```
path      db      "textfile.asc",0
msg1      db      "File loaded into allocated memory block.",
                   ODH,0AH
msg2      db      "Allocated memory now being freed
                   (deallocated).",ODH,0AH
handle    dw      ?
mem_seg   dw      ?
file_len  dw      ?
;
begin:    open_handle  path,0
          jc       error_open     ;Routine not shown
          mov      handle,ax      ;Save handle
          move_ptr handle,0,0,2   ;See Function 42H
          jc       error_move     ;Routine not shown
          mov      file_len,ax    ;Save file length
          set_block  last_inst    ;See Function 4AH
          jc       error_setblk   ;Routine not shown
          allocate_memory  file_len ;THIS FUNCTION
          jc       error_alloc    ;Routine not shown
          mov      mem_seg,ax     ;Save address of new memory
          move_ptr handle,0,0,0   ;See Function 42H
          jc       error_move     ;Routine not shown
          push     ds             ;Save DS
          mov      ax,mem_seg     ;Get segment of new memory
          mov      ds,ax          ;Point DS at new memory
          read_handle  cs:handle,0,cs:file_len ;Read file into
;                                                new memory
          pop      ds             ;Restore DS
          jc       error_read     ;Routine not shown
;         (CODE TO PROCESS FILE GOES HERE)
          write_handle stdout,msg1,42 ;See Function 40H
          jc       write_error    ;Routine not shown
          free_memory  mem_seg    ;See Function 49H
          jc       error_freemem ;Routine not shown
          write_handle stdout,msg2,49 ;See Function 40H
          jc       write_error    ;Routine not shown
```

Free Allocated Memory (Function 49H)



Call
AH = 49H
ES
    Segment address of memory to be freed

Return
Carry set:
AX
    7 = Memory control blocks damaged
    9 = Incorrect segment
Carry not set:
    No error

Function 49H releases (makes available) a block of memory previously allocated with Function 48H (Allocate Memory). ES must contain the segment address of the memory block to be released.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
7	Memory control blocks damaged (a user program changed memory that doesn't belong to it).
9	The memory pointed to by ES was not allocated with Function 48H.

Macro Definition:

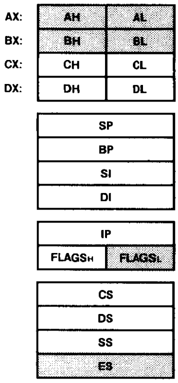
```
free_memory  macro   seg_addr
             mov     ax,seg_addr
             mov     es,ax
             mov     ah,49H
             int     21H
             endm
```

Example

The following program opens the file named TEXTFILE.ASC, calculates its size with Move File Pointer (42H), allocates a block of memory the size of the file, reads the file into the allocated memory block, then frees the allocated memory.

```
path        db      "textfile.asc",0
msg1        db      "File loaded into allocated memory block.",
                     0DH,0AH
msg2        db      "Allocated memory now being freed
                     (deallocated).",0DH,0AH
handle      dw       ?
mem_seg     dw       ?
file_len    dw       ?
;
begin:      open_handle  path,0
            jc       error_open      ;Routine not shown
            mov      handle,ax       ;Save handle
            move_ptr handle,0,0,2    ;See Function 42H
            jc       error_move      ;Routine not shown
            mov      file_len,ax     ;Save file length
            set_block  last_inst     ;See Function 4AH
            jc       error_setblk    ;Routine not shown
            allocate_memory  file_len ;See Function 48H
            jc       error_alloc     ;Routine not shown
            mov      mem_seg,ax      ;Save address of new memory
            mov_ptr  handle,0,0,0    ;See Function 42H
            jc       error_move      ;Routine not shown
            push     ds              ;Save DS
            mov      ax,mem_seg      ;Get segment of new memory
            mov      ds,ax           ;Point DS at new memory
            read_handle  handle,code,file_len ;Read file into
;                                                 new memory
            pop      ds              ;Restore DS
            jc       error_read      ;Routine not shown
;           (CODE TO PROCESS FILE GOES HERE)
            write_handle stdout,msg1,42 ;See Function 40H
            jc       write_error     ;Routine not shown
            free_memory  mem_seg     ;THIS FUNCTION
            jc       error_freemem   ;Routine not shown
            write_handle stdout,msg2,49 ;See Function 40H
            jc       write_error     ;Routine not shown
```

Set Block (Function 4AH)



Call
AH = 4AH
BX
    Paragraphs of memory
ES
    Segment address of memory area

Return
Carry set:
AX
    7 = Memory control blocks damaged
    8 = Insufficient memory
    9 = Incorrect segment
BX
    Paragraphs of memory available
Carry not set:
    No error

Function 4AH changes the size of a memory allocation block. ES must contain the segment address of the memory block. BX must contain the new size of the memory block, in paragraphs (1 paragraph is 16 bytes).

MS-DOS attempts to change the size of the memory block. If the call fails on a request to increase memory, BX returns the maximum size (in paragraphs) to which the block can be increased.

Because MS-DOS allocates all of available memory to a .COM program, this call is most often used to reduce the size of a program's initial memory allocation block.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
7	Memory control blocks destroyed (a user program changed memory that doesn't belong to it).
8	Not enough free memory to satisfy the request.
9	Wrong address in ES (the memory block it points to cannot be modified with Set Block).

Macro Definition:

This macro is set up to shrink the initial memory allocation block of a .COM program. It takes as a parameter the offset of the first byte following the last instruction of a program (LASTINST in the sample programs), uses it to calculate the number of paragraphs in the program, then adds 17 to the result -- 1 to round up and 16 to set aside 256 bytes for a stack. It then sets up SP and BP to point to this stack.

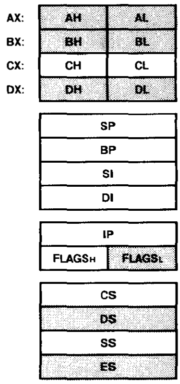
```
set_block  macro   last_byte
                   mov     bx,offset last_byte
                   mov     cl,4
                   shr     bx,cl
                   add     bx,17
                   mov     ah,4AH
                   int     21H
                   mov     ax,bx
                   shl     ax,cl
                   dec     ax
                   dec     ax
                   mov     sp,ax
                   endm
```

Example

The following program invokes a second copy of COMMAND.COM and executes a Dir (directory) command.

```
pgm_file  db      "command.com",0
cmd_line  db       9,"/c dir /w",0DH
parm_blk  db       14 dup (?)
reg_save  db       10 dup (?)
;
begin: set_block  last_inst                   ;THIS FUNCTION
       exec       pgm_file,cmd_line,parm_blk,0 ;See Function
                                                ;4BH
```

Load and Execute Program (Function 4BH, Code 00H)



Call
AH = 4BH
AL = 00H
DS:DX
Pointer to pathname
ES:BX
Pointer to parameter block

Return
Carry set:
AX
1 = Invalid function
2 = File not found
8 = Insufficient memory
10 = Bad environment
11 = Bad format
Carry not set:
No error

Function 4BH, Code 00H loads and executes a program. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the drive and pathname of an executable program file. BX must contain the offset (from the segment address in ES) of a parameter block. AL must contain 0.

There must be enough free memory for MS-DOS to load the program file. All available memory is allocated to a program when it is loaded, so you must free some memory with Function 4AH (Set Block) before using this function request to load and execute another program. Unless memory is needed for some other purpose, shrink to the minimum amount of memory required by the current process before issuing this function request.

MS-DOS creates a Program Segment Prefix for the program being loaded, and sets the terminate and Control-C addresses to the instruction that immediately follows the call to Function 4BH in the invoking program.

The parameter block consists of four addresses:

Offset (Hex)	Length (Bytes)	Description
00	2 (word)	Segment address of environment to be passed; 00H means copy the parent's environment.
02	4 (dword)	Segment:Offset of command line to be placed at offset 80H of the new Program Segment Prefix. This must be a correctly formed command line no longer than 128 bytes.
06	4 (dword)	Segment:Offset of FCB to be placed at offset 5CH of the new Program Segment Prefix (the Program Segment Prefix is described in Chapter 4).
0A	4 (dword)	Segment:Offset of FCB to be placed at offset 6CH of the new Program Segment Prefix.

All open files of a program are available to the newly loaded program, giving the parent program control over the definition of standard input, output, auxiliary, and printer devices. For example, a program could write a series of records to a file, open the file as standard input, open a second file as standard output, then use Load and Execute Program to load and execute a program that takes its input from standard input, sorts records, and writes to standard output.

The loaded program also receives an environment, a series of ASCIZ strings of the form parameter=value (for example, VERIFY=ON). The environment must begin on a paragraph boundary, be less than 32K bytes long, and end with a byte of 00H (that is, the final entry consists of an ASCII string followed by two bytes of 00H). After the last byte of zeros is a set of initial arguments passed to a program that contains a word count followed by an ASCIZ string. If the file is found in the current directory, the ASCIZ string contains the drive and pathname of the executable program as passed to Function 4BH. If the file is found in the path, the filename is concatenated with the path information. (A program may use this area to determine where the program was loaded from.) If the word environment address is 0, the loaded program either inherits a copy of the parent's environment or receives a new environment built for it by the parent.

Place the segment address of the environment at offset 2CH of the new Program Segment Prefix. To build an environment for the loaded program, put it on a paragraph boundary and place the segment address of the environment in the first word of the parameter block. To pass a copy of the parent's environment to the loaded program, put 00H in the first word of the parameter block.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL is not 0 or 3.
2	Program file not found or path is invalid.
8	Not enough memory to load the program.
11	Program file is an .EXE file that contains internally inconsistent information.

Executing Another Copy of COMMAND.COM

Because COMMAND.COM takes care of such details as building pathnames, searching the command path for program files, and relocating .EXE files, the simplest way to load and execute another program is to load and execute an additional copy of COMMAND.COM, passing it a command line that includes the /C switch -- which tells COMMAND.COM to treat the remainder of the command line as an executable command -- that invokes the .COM or .EXE file.

This requires 17K bytes of available memory, so a program that does this should be sure to shrink its initial memory allocation block with Function 4AH (Set Block). The format of a command line that contains the /C switch:

<length>/C <command><0DH>

<Length> is the length of the command line, counting the length byte but not counting the ending carriage return (0DH).

<Command> is any valid MS-DOS command.

<0DH> is a carriage return character.

If a program executes another program directly -- naming it as the program file to Function 4BH instead of COMMAND.COM -- it must perform all the processing normally done by COMMAND.COM.

Macro Definition:

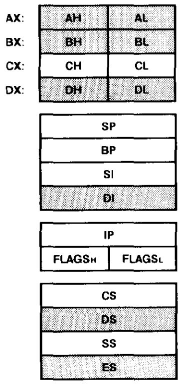
```
exec    macro   path,command,parms
        mov     dx,offset path
        mov     bx,offset parms
        mov     word ptr parms[02H],offset command
        mov     word ptr parms[04H],cs
        mov     word ptr parms[06H],5CH
        mov     word ptr parms[08H],es
        mov     word ptr parms[0AH],6CH
        mov     word ptr parms[0CH],es
        mov     al,0
        mov     ah,4BH
        int     21H
        endm
```

Example

The following program invokes a second copy of COMMAND.COM and executes a Dir (directory) command with the /W (wide) switch:

```
pgm_file  db      "command.com",0
cmd_line  db       9,"/c dir /w",0DH
parm_blk  db       14 dup (?)
reg_save  db       10 dup (?)
;
begin:
   set_block  last_inst                       ;See Function 4AH
   exec       pgm_file,cmd_line,parm_blk,0 ;THIS FUNCTION
```

Load Overlay(Function 4BH, Code 03H)



Call
AH = 4BH
AL = 03H
DS:DX
    Pointer to pathname
ES:BX
    Pointer to parameter block

Return
Carry set:
AX
    1 = Invalid function
    2 = File not found
    8 = Insufficient memory
    10 = Bad environment
Carry not set:
    No error

Function 4BH, Code 03H loads a program segment (overlay). DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the drive and pathname of the program file. BX must contain the offset (from the segment address in ES) of a parameter block. AL must contain 3.

MS-DOS assumes that the invoking program is loading into its own address space, so no free memory is required. A Program Segment Prefix is not created.

The parameter block is four bytes long:

Offset (Hex)	Length (Bytes)	Description
00	2 (word)	Segment address where program is to be loaded.
02	2 (word)	Relocation factor. This is usually the same as first word of the parameter block; for a description of an .EXE file and relocation, see Chapter 5).

If there is an error, the carry flag (CF) is set and the error code is returned in AX.

Code	Meaning
1	AL is not 00H or 03H.
2	Program file not found or path is invalid.
8	Not enough memory to load the program.

Macro Definition:

```
exec_ovl  macro   path,parms,seg_addr
          mov     dx,offset path
          mov     bx,offset parms
          mov     parms,seg_addr
          mov     parms[02H],seg_addr
          mov     al,3
          mov     ah,4BH
          int     21H
          endm
```

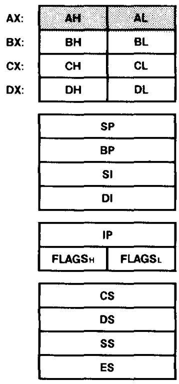
Example

The following program opens a file named TEXTFILE.ASC, redirects standard input to that file, loads MORE.COM as an overlay, and calls MORE.COM. MORE.COM reads TEXTFILE.ASC as standard input.

```
stdin     equu      0
;
file      db     "TEXTFILE.ASC",0
cmd_file  db     "\more.com",0
parm_blk  dw      4 dup (?)
handle    dw     ?
new_mem   dw     ?
;
begin:    set_block    last_inst          ;see Function 4AH
          jc           setblock_error     ;routine not shown
          allocate_memory  2000           ;see Function 48H
          jc           allocate_error     ;routine not shown
          mov          new_mem,ax         ;save seg of memory
          open_handle  file,0             ;see Function 3DH
          jc           open_error         ;routine not shown
          mov          handle,ax          ;save handle
          xdup2        handle,stdin       ;see Function 45H
          jc           dup2_error         ;routine not shown
          close_handle handle             ;see Function 3EH
          jc           close_error        ;routine not shown
          mov          ax,new_mem         ;addr of new memory
```

```
        exec_ovl cmd_file,parm_blk,ax     ;THIS FUNCTION
        jc          exec_error          ;routine not shown
        mov         ax,new_mem          ;point to overlay
        sub         ax,10h              ;no PSP for overlay
        mov         ds,ax               ;DS for overlay
        call        cs:overlay          ;call the overlay
        push        cs                  ;restore DS to
        pop         ds                  ;original segment
        free_memory new_mem             ;see Function 49H
        jc          free_error          ;routine not shown
;
```

End Process (Function 4CH)



Call
AH = 4CH
AL
Return code

Return
None

Function 4CH terminates a process and returns to MS-DOS. AL contains a return code that can be retrieved by the parent process with Function 4DH (Get Return Code of Child Process) or the If command using ERRORLEVEL.

MS-DOS closes all open handles, ends the current process, and returns control to the invoking process.

This function request doesn't require that CS contain the segment address of the Program Segment Prefix. You should use it to end a program (rather than Interrupt 20H or a jump to location 0) unless it is absolutely imperative that your program be compatible with pre-2.0 versions of MS-DOS.

Macro Definition:

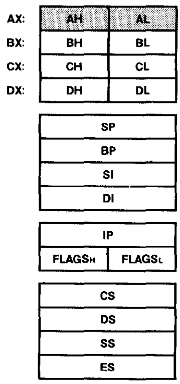
```
end_process macro return_code
            mov   al,return_code
            mov   ah,4CH
            int   21H
            endm
```

Example

The following program displays a message and returns to MS-DOS with a return code of 8. It uses only the opening portion of the sample program skeleton shown at the beginning of this chapter.

```
message   db      "Displayed by FUNC_4CH example",0DH,0AH,"$"
;
begin:    display      message    ;See Function 09H
          end_process  8          ;THIS FUNCTION
code      ends
          end          code
```

Get Return Code of Child Process (Function 4DH)



Call
AH = 4DH

Return
AX
    Return code

Function 4DH retrieves the return code specified when a child process terminated with either Function 31H (Keep Process) or Function 4CH (End Process). The code is returned in AL. AH returns a code that specifies the reason the program ended:

Code	Meaning
0	Normal termination.
1	Terminated by Control-C.
2	Critical device error.
3	Function 31H (Keep Process).

The exit code can be retrieved only once.

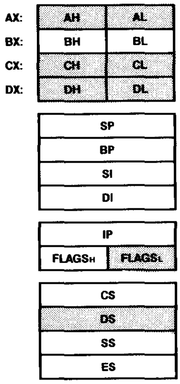
Macro Definition:

```
ret_code  macro
          mov    ah,4DH
          int    21H
          endm
```

Example

Because the meaning of a return code varies, no example is included for this function request.

Find First File (Function 4EH)



Call
AH = 4EH
DS:DX
   Pointer to pathname
CX
   Attributes to match

Return
Carry set:
AX
   2 = File not found
  18 = No more files
Carry not set:
   No error

Function 4EH searches the specified or current directory for the first entry that matches the specified pathname. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname that can include wildcard characters. CX must contain the attribute to be used in searching for the file, as described in Section 1.5.6, "File Attributes," earlier in this chapter.

If the attribute field is hidden file, system file, or directory entry (02H, 04H, or 10H), or any combination of those values, all normal file entries are also searched. To search all directory entries except the volume label, set the attribute byte to 16H (hidden file and system file and directory entry).

If a directory entry is found that matches the name and attribute, the current DTA is filled as follows:

Offset	Length	Description
00H	21	Reserved for subsequent Find Next File (Function Request 4FH).
15H	1	Attribute found.
16H	2	Time file was last written.
18H	2	Date file was last written.

1AH	2	Low word of file size.
1CH	2	High word of file size.
1EH	13	Name and extension of the file, followed by 00H. All blanks are removed; if there is an extension, it is preceded by a period (it appears just as you would enter it in a command).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
2	The specified path is invalid or doesn't exist.
18	No matching directory entry was found.

Macro Definition:

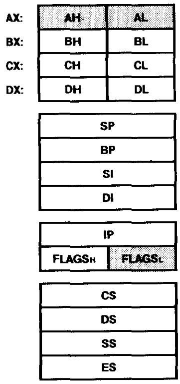
```
find_first_file  macro  path,attrib
                 mov    dx,offset path
                 mov    cx,attrib
                 mov    ah,4EH
                 int    21H
                 endm
```

Example

The following program displays a message that specifies whether a file named REPORT.ASM exists in the current directory on the disk in drive B.

```
yes         db         "FILE EXISTS.",0DH,0AH,"$"
no          db         "FILE DOES NOT EXIST.",0DH,0AH,"$"
path        db         "b:report.asm",0
buffer      db          43 dup (?)
;
begin:      set_dta    buffer               ;See Function 1AH
            find_first_file  path,0         ;THIS FUNCTION
            jc         error_findfirst      ;Routine not shown
            cmp        al,12H               ;File found?
            je         not_there            ;No
            display    yes                  ;See Function 09H
            jmp        return               ;All done
not_there:  display    no                   ;See Function 09H
```

Find Next File (Function 4FH)



Call
AH = 4FH

Return
Carry set:
AX
18 = No more files
Carry not set:
No error

Function 4FH searches for the next directory entry that matches the name and attributes specified in a previous Function 4EH (Find First File). The current DTA must contain the information filled in by Function 4EH (Find First File).

If a matching entry is found, the current DTA is filled just as it was for Find First File (see the previous function request description).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
2	The specified path is invalid or doesn't exist.
18	No matching directory entry was found.

Macro Definition:

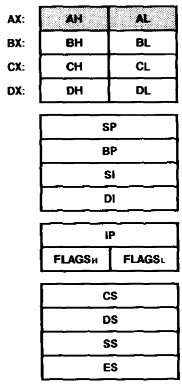
```
find_next_file  macro
                mov     ah,4FH
                int     21H
                endm
```

Example

The following program displays the number of files in the current directory on the disk in drive B.

```
message      db        "No files",0DH,0AH,"$"
files        dw         ?
path         db        "b:*.*",0
buffer       db         43 dup (?)
;
begin:       set_dta   buffer             ;See Function 1AH
             find_first_file  path,0      ;See Function 4EH
             jc        error_findfirst    ;Routine not shown
             cmp       al,12H             ;Directory empty?
             je        all_done           ;Yes, go home
             inc       files              ;No, bump file counter
search_dir:  find_next_file               ;THIS FUNCTION
             jc        error_findnext     ;Routine not shown
             cmp       al,12H             ;Any more entries?
             je        done               ;No, go home
             inc       files              ;Yes, bump file counter
             jmp       search_dir         ;And check again
done:        convert   files,10,message   ;See end of chapter
all_done:    display   message            ;See Function 09H
```

Get Verify State (Function 54H)



Call
AH = 54H

Return
AL
0 = No verify after write
1 = Verify after write

Function 54H checks whether MS-DOS verifies write operations to disk files. The status is returned in AL: 0 if verify is off, 1 if verify is on.

You can set the verify status with Function 2EH (Set/Reset Verify Flag).

Macro Definition:

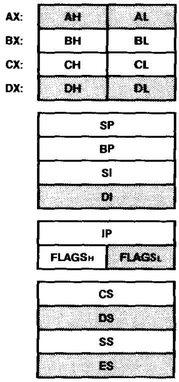
```
get_verify  macro
            mov     ah,54H
            int     21H
            endm
```

Example

The following program displays the verify status:

```
message     db      "Verify ","$"
on          db      "on.",0DH,0AH,"$"
off         db      "off.",0DH,0AH,"$"
;
begin:      display message         ;See Function 09H
            get_verify              ;THIS FUNCTION
            cmp     al,0            ;Is flag off?
            jg      ver_on          ;No, it's on
            display off             ;See Function 09H
            jmp     return          ;Go home
ver_on:     display on              ;See Function 09H
```

Change Directory Entry (Function 56H)



Call
AH = 56H
DS:DX
    Pointer to pathname
ES:DI
    Pointer to second pathname

Return
Carry set:
AX
    2 = File not found
    5 = Access denied
   17 = Not same device
Carry not set:
    No error

Function 56H renames a file by changing its directory entry. DX must contain the offset (from the segment address in DS) of an ASCIZ string that contains the pathname of the entry to be changed. DI must contain the offset (from the segment address in ES) of an ASCIZ string that contains a second pathname to which the first is to be changed.

If a directory entry for the first pathname exists, it is changed to the second pathname.

The directory paths need not be the same; in effect, you can move the file to another directory by renaming it. You cannot use this function request to copy a file to another drive, however: if the second pathname specifies a drive, the first pathname must specify or default to the same drive.

This function request cannot be used to rename a hidden file, system file, or subdirectory. If there is an error, the carry flag (CF) is set and the error code is returned in AX.

Code	Meaning
2	One of the paths is invalid or not open.
5	The first pathname specifies a directory, the second pathname specifies an existing file, or the second directory entry could not be opened.
17	Both files are not on the same drive.

Macro Definition:

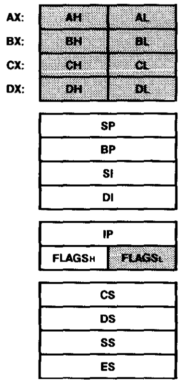
```
rename_file  macro  old_path,new_path
             mov    dx,offset old_pat
             push   ds
             pop    es
             mov    di,offset new_path
             mov    ah,56H
             int    21H
             endm
```

Example

The following program prompts for the name of a file and a new name, then renames the file.

```
prompt1   db      "Filename: $"
prompt2   db      "New name: $"
old_path  db       15,?,15 dup (?)
new_path  db       15,?,15 dup (?)
crlf      db       0DH,0AH,"$"
;
begin:    display prompt1             ;See Function 09H
          get_string  15,old_path     ;See Function 0AH
          xor     bx,bx               ;To use BL as index
          mov     bl,old_path[1]      ;Get string length
          mov     old_path[bx+2],0    ;Make an ASCIZ string
          display crlf                ;See Function 09H
          display prompt2             ;See Function 09H
          get_string  15,new_path     ;See Function 0AH
          xor     bx,bx               ;To use BL as index
          mov     bl,new_path[1]      ;Get string length
          mov     new_path[bx+2],0    ;Make an ASCIZ string
          display crlf                ;See Function 09H
          rename_file old_path[2],new_path[2];THIS FUNCTION
          jc      error_rename        ;Routine not shown
```

Get/Set Date/Time of File(Function 57H)



Call
AH = 57H
AL = Function code
   0 = Get date and time
   1 = Set date and time
BX
   Handle
CX (if AL=1)
   Time to be set
DX (if AL=1)
   Date to be set

Return
Carry set:
AX
   1 = Invalid function
   6 = Invalid handle
Carry not set:
CX (if AL=0)
   Time file last written
DX (if AL=0)
   Date file last written

Function 57H gets or sets the time and date a file was last written. To get the time and date, AL must contain 0; the time and date are returned in CX and DX. To set the time and date, AL must contain 1; CX and DX must contain the time and date. BX must contain the file handle. The time and date are in the form described in "Fields of the FCB" in Section 1.8.1.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	AL is not 0 or 1.
6	The handle in BX is invalid or not open.

Macro Definition:

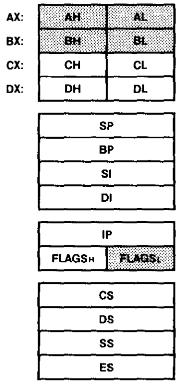
```
get_set_date_time macro   handle,action,time,date
                  mov     bx,handle
                  mov     al,action
                  mov     cx,word ptr time
                  mov     dx,word ptr date
                  mov     ah,57H
                  int     21H
                  endm
```

Example

The following program gets the date of the file named REPORT.ASM in the current directory on the disk in drive B, increments the day, increments the month or year if necessary, and sets the new date of the file.

```
month      db      31,28,31,30,31,30,31,31,30,31,30,31
path       db      "b:report.asm",0
handle     dw      ?
time       db      2 dup (?)
date       db      2 dup (?)
;
begin:     open_handle  path,0          ;See Function 3DH
           mov     handle,ax            ;Save handle
           get_set_date_time handle,0,time,date;THISFUNCTION
           jc      error_time           ;Routine not shown
           mov     word ptr time,cx     ;Save time
           mov     word ptr date,dx     ;Save date
           convert_date  date[-24]      ;See end of chapter
           inc     dh                   ;Increment day
           xor     bx,bx                ;To use BL as index
           mov     bl,dl                ;Get month
           cmp     dh,month[bx-1]       ;Past last day?
           jle     month_ok             ;No, go home
           mov     dh,1                 ;Yes, set day to 1
           inc     dl                   ;Increment month
           cmp     dl,12                ;Is it past December?
           jle     month_ok             ;No, go home
           mov     dl,1                 ;Yes, set month to 1
           inc     cx                   ;Increment year
month_ok:  pack_date  date              ;See end of chapter
           get_set_date_time handle,1,time,date;THISFUNCTION
           jc      error_time           ;Routine not shown
           close_handle  handle         ;See Function 3EH
           jc      error_close          ;Routine not shown
```

Get/Set Allocation Strategy (Function 58H)



Call
AH = 58H
AL
  0 = Get strategy
  1 = Set strategy
BX (AL=1)
  0 = First fit
  1 = Best fit
  2 = Last fit

Return
Carry set:
AX
  1 = Invalid function code
Carry not set:
AX (AL=0)
  0 = First fit
  1 = Best fit
  2 = Last fit

Function 58H gets or sets the strategy used by MS-DOS to allocate memory when requested by a process. If AL contains 0, the strategy is returned in AX. If AL contains 1, BX must contain the strategy. The three possible strategies are:

Value	Name	Description
0	First fit	MS-DOS starts searching at the lowest available block and allocates the first block it finds (the allocated memory is the lowest available block). This is the default strategy.
1	Best fit	MS-DOS searches each available block and allocates the smallest available block that satisfies the request.
2	Last fit	MS-DOS starts searching at the highest available block and allocates the first block it finds (the allocated memory is the highest available block).

You can use this function request to control how MS-DOS uses its memory resources.

If there is an error, the carry flag (CF) is set and the error code is returned in AX.

Code	Meaning
1	AL doesn't contain 0 or 1, or BX doesn't contain 0, 1, or 2.

Macro Definition:

```
alloc_strat macro   code,strategy
            mov     bx,strategy
            mov     al,code
            mov     ah,58H
            int     21H
            endm
```

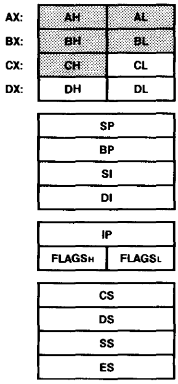
Example

The following program displays the memory allocation strategy in effect, then forces subsequent memory allocations to the top of memory by setting the strategy to last fit (code 2).

```
get       equu      0
```

```
get       equ       0
set       equ       1
stdout    equ       1
last_fit  equ       2
;
first     db        "First fit      ",0DH,0AH
best      db        "Best fit       ",0DH,0AH
last      db        "Last fit       ",0DH,0AH
;
begin:    alloc_strat get                ;THIS FUNCTION
          jc        alloc_error          ;routine not shown
          mov       cl,4                 ;multiply code by 16
          shl       ax,cl                ;to calculate offset
          mov       dx,offset first      ;point to first msg
          add       dx,ax                ;add to base address
          mov       bx,stdout            ;handle for write
          mov       cs,16                ;write 16 bytes
          mov       ah,40h               ;write handle
          int       21H                  ;system call
;         jc        write_error          ;routine not shown
          alloc_strat set,last_fit       ;THIS FUNCTION
;         jc        alloc_error          ;routine not shown
```

Get Extended Error (Function 59H)



Call
AH = 59H
BX = 0

Return
AX
  Extended error code
BH
  Error class (see text)
BL
  Suggested action (see text)
CH
  Locus (see text)

CL, DX, SI, DI, BP, DS, ES destroyed

Function 59H retrieves an extended error code for the immediately previous system call. Each release of MS-DOS extends the error codes to cover new capabilities. These new codes are mapped to a simpler set of error codes based on Version 2.0 of DOS, so that existing programs can continue to operate correctly. Note that all registers except CS:IP and SS:SP are destroyed by this call.

A user-written Interrupt 24H handler can use Function 59H (Get Extended Error) to get detailed information about the error that caused the interrupt to be issued.

The input BX is a version indicator which says what level of error handling the application was written for. The current level is 0.

The extended error code consists of four separate codes in AX, BH, BL, and CH that give as much detail as possible about the error and suggest how the issuing program should respond.

BH -- Error Class

BH returns a code that describes the class of error that occurred:

Class	Description
1	Out of a resource, such as storage or channels.
2	Not an error, but a temporary situation (such as a locked region in a file) that can be expected to end.
3	Authorization problem.
4	An internal error in system software.
5	Hardware failure.
6	A system software failure not the fault of the active process (could be caused by missing or incorrect configuration files, for example).
7	Application program error.
8	File or item not found.
9	File or item of invalid format, type, or otherwise invalid or unsuitable.
10	File or item interlocked.
11	Wrong disk in drive, bad spot on disk, or other problem with storage medium.
12	Other error.

BL -- Suggested Action

BL returns a code that suggests how the issuing program can respond to the error:

Action	Description
1	Retry, then prompt user.
2	Retry after a Pause.
3	If the user entered data such as a drive letter or file name, prompt for it again.
4	Terminate with cleanup.
5	Terminate immediately. The system is so unhealthy that the program should exit as soon as possible

without taking the time to close files and update indexes.

6 Error is informational.

7 Prompt the user to perform some action, such as changing disks, then retry the operation.

CH -- Locus

CH returns a code that provides additional information to help locate the area involved in the failure. This code is particularly useful for hardware failures (BH=5).

Locus	Description
1	Unknown.
2	Related to random access block devices, such as a disk drive.
3	Related to Network.
4	Related to serial access character devices, such as a printer.
5	Related to random access memory.

Your programs should handle errors by noting the error return from the original system call, then issuing this system call to get the extended error code. If the program does not recognize the extended error code, it should respond to the original error code.

This system call is available during Interrupt 24H and may be used to return network-related errors.

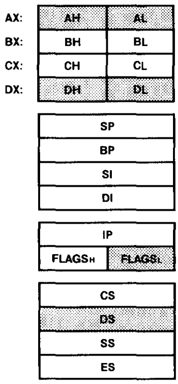
Macro Definition:

```
get_error macro
          mov    ah, 59H
          int    21H
          endm
```

Example

Because so much detail is provided by this function request, an example is not shown. User programs can interpret the various codes to determine what sort of messages or prompts should be displayed, what action to take, and whether to terminate the program if recovery from the errors isn't possible.

Create Temporary File (Function 5AH)



Call
AH = 5AH
CX
    Attribute
DS:DX
    Pointer to pathname followed by a byte of 0 and 13 bytes of memory

Return
Carry set:
AX
    3 = Path not found
    5 = Access denied
Carry not set:
AX
    Handle

Function 5AH creates a file with a unique name. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies a pathname and 13 bytes of memory (to hold the filename). CX must contain the attribute to be assigned to the file, as described in Section 1.5.6, "File Attributes," earlier in this chapter.

MS-DOS creates a unique filename and appends it to the pathname pointed to by DS:DX, creates the file and opens it in compatibility mode, then returns the file handle in AX. A program that needs a temporary file should use this function request to avoid name conflicts.

MS-DOS does <u>not</u> automatically delete a file created with Function 5AH when the creating process exits. When the file is no longer needed, it should be deleted.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
3	The directory pointed to by DS:DX is invalid or doesn't exist.
5	Access denied.

Macro Definition:
```
create_temp macro  pathname,attrib
            mov    cx,attrib
            mov    dx,offset pathname
            mov    ah,5AH
            int    21H
            endm
```

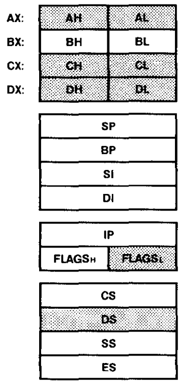
Example

The following program creates a temporary file in the directory named \WP\DOCS, copies a file in the current directory named TEXTFILE.ASC into the temporary file, then closes both files.

```
stdout    equu      1
;
file      db        "TEXTFILE.ASC",0
path      db        "\WP\DOCS",0
temp      db         13 dup (0)
open_msg  db        " opened.",0DH,0AH
crl_msg   db        " created.",0DH,0AH
rd_msg    db        " read into buffer.",0DH,0AH
wr_msg    db        "Buffer written to "
cl_msg    db        "Files closed.",0DH,0AH
crlf      db        0DH,0AH
handle1   dw         ?
handle2   dw         ?
buffer    db         512 dup (?)
;
begin:    open_handle file,0              ;see Function 3DH
          jc         open_error           ;routine not shown
          mov        handle1,ax           ;save handle
          write_handle stdout,file,12     ;see Function 40H
          jc         write_error          ;routine not shown
          write_handle stdout,open_msg,10 ;see Function 40H
          jc         write_error          ;routine not shown
          create_temp path,0              ;THIS FUNCTION
          jc         create_error         ;routine not shown
          mov        handle2,ax           ;save handle
          write_handle stdout,path,8      ;see Function 40H
          jc         write_error          ;routine not shown
          display_char "\"                ;see Function 02H
          write_handle stdout,temp,12     ;see Function 40H
          jc         write_error          ;routine not shown
          write_handle stdout,crl_msg,11 ;See Function 40H
          jc         write_error          ;routine not shown
          read_handle handle1,buffer,512  ;see Function 3FH
          jc        read_error            ;routine not shown
          write_handle stdout,file,12     ;see Function 40H
          jc        write_error           ;routine not shown
          write_handle stdout,rd_msg,20 ;see Function 40H
          jc        write_error           ;routine not shown
          write_handle handle2,buffer,512 ;see Function 40H
          jc        write_error           ;routine not shown
          write_handle stdout,wr_msg,18 ;see Function 40H
          jc         write_error          ;routine not shown
```

```
write_handle stdout,temp,12   ;see Function 40H
jc          write_error       ;routine not shown
write_handle stdout,crlf,2    ;see Function 40H
jc          write_error       ;routine not shown
close_handle handle1          ;see Function 3EH
jc          close_error       ;routine not shown
close_handle handle2          ;see Function 3EH
jc          close_error       ;routine not shown
write_handle stdout,cl_msg,15 ;see Function 40H
jc          write_error       ;routine not shown
```

Create New File (Function 5BH)



Call
AH = 5BH
CX
    Attribute
DS:DX
    Pointer to pathname

Return
Carry set:
AX
     3 = Path not found
     4 = Too many open files
     5 = Access denied
    80 = File already exists
Carry not set:
AX
    Handle

Function 5BH creates a new file. DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies a pathname. CX contains the attribute to be assigned to the file, as described in Section 1.5.6, "File Attributes."

If there is no existing file with the same filename, MS-DOS creates the file, opens it in compatibility mode, and returns the file handle in AX.

Unlike Function 3CH (Create Handle), this function request fails if the specified file exists, rather than truncating it to a length of 0. The existence of a file is used as a semaphore in a multitasking system; you can use this system call as a test-and-set semaphore.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
3	The directory pointed to by DS:DX is invalid or doesn't exist.
4	No free handles are available in the current process, or the internal system tables are full.
5	Access denied.
80	A file with the same specification pointed to by DS:DX already exists.

Macro Definition:

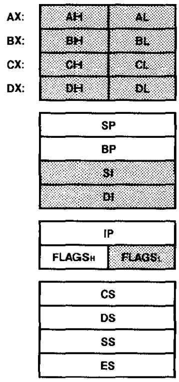
```
create_new macro  pathname,attrib
           mov    cx, attrib
           mov    dx, offset pathname
           mov    ah, 5BH
           int    21H
           endm
```

Example

The following program attempts to create a new file in the current directory named REPORT.ASM. If the file already exists, the program displays an error message and returns to MS-DOS. If the file doesn't exist and there are no other errors, the program saves the handle and continues processing.

```
err_msg  db       "FILE ALREADY EXISTS",0DH,0AH,"$"
path     db       "REPORT.ASM",0
handle   dw        ?
;
begin:     create_new path,0             ;THIS FUNCTION
           jnc        continue           ;further processing
           cmp        ax,80              ;file already exist?
           jne        error              ;routine not shown
           display    err_msg            ;see Function 09H
           jmp        return             ;return to MS-DOS
continue:  mov        handle,ax          ;save handle
;
;          (further processing here)
```

Lock (Function 5CH, Code 00H)



Call
AH = 5CH
AL = 00H
BX
   Handle
CX:DX
   Offset of region to be locked
SI:DI
   Length of region to be locked

Return
Carry set:
AX
   1 = Invalid function code
   6 = Invalid handle
   22 = Lock violation
Carry not set:
   No error

Function 5CH, Code 00H denies all access (read or write) by any other process to the specified region of the file. BX must contain the handle of the file that contains the region to be locked. CX:DX (a 4-byte integer) must contain the offset in the file of the beginning of the region. SI:DI (a 4-byte integer) must contain the length of the region.

If another process attempts to use (read or write) a locked region, MS-DOS retries three times; if the retries fail, MS-DOS issues Interrupt 24H for the requesting process. You can change the number of retries with Function 44H, Code 0BH (IOCTL Retry).

The locked region can be anywhere in the file. Locking beyond the end of the file is not an error. A region should be locked for a brief period; it should be considered an error if a region is locked for more than 10 seconds.

Function 45H (Duplicate File Handle) and Function 46H (Force Duplicate File Handle) duplicate access to any locked region. Passing an open file to a child process with Function 4BH, Code 00H (Load and Execute Program) does not duplicate access to locked regions.

If a program closes a file that contains a locked region or terminates with an open file that contains a locked region, the result is undefined. Programs that might be terminated by Interrupt 23H (Control-C) or Interrupt 24H (a fatal error) should trap these interrupts and unlock any locked regions before exiting.

Programs should not rely on being denied access to a locked region; a program can determine the status of a region (locked or unlocked) by attempting to lock the region and examining the error code.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	File sharing must be loaded to use this function request.
6	The handle in BX is not a valid, open handle.
33	All or part of the specified region is already locked.

Macro Definition:

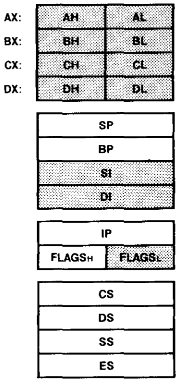
```
lock    macro   handle,start,bytes
        mov     bx, handle
        mov     cx, word ptr start
        mov     dx, word ptr start+2
        mov     si, word ptr bytes
        mov     di, word ptr bytes+2
        mov     al, 0
        mov     ah, 5CH
        int     21H
        endm
```

Example

The following program opens a file named FINALRPT in Deny None mode and locks two portions of it: the first 128 bytes and bytes 1024 through 5119. After some (unspecified) processing, it unlocks the same portions and closes the file.

```
stdout     equ          1
;
start1     dd           0
lgth1      dd           128
start2     dd           1023
lgth2      dd           4096
file       db           "FINALRPT",0
op_msg     db           " opened.",0DH,0AH
l1_msg     db           "First 128 bytes locked.",0DH,0AH
l2_msg     db           "Bytes 1024-5119 locked.",0DH,0AH
u1_msg     db           "First 128 bytes unlocked.",0DH,0AH
u2_msg     db           "Bytes 1024-5119 unlocked.",0DH,0AH
cl_msg     db           " closed.:,0DH,0AH
handle     dw           ?
;
begin:     open_handle file,01000010b     ;see Function 3DH
           jc          open_error         ;routine not shown
           write_handle stdout,file,8     ;see Function 40H
           jc          write_error        ;routine not shown
           write_handle stdout,op_msg,10  ;see Function 40H
           jc          write_error        ;routine not shown
           mov         handle,ax          ;save handle
           lock        handle,start1,lgth1 ;THIS FUNCTION
           jc          lock_error         ;routine not shown
           write_handle stdout,l1_msg,25  ;see Function 40H
           jc          write_error        ;routine not shown
           lock        handle,start2,lgth2 ;THIS FUNCTION
           jc          lock_error         ;routine not shown
           write_handle stdout,l2_msg,25  ;see Function 40H
           jc          write_error        ;routine not shown
;
; ( Further processing here )
;
           unlock      handle,start1,lgth1 ;See Function 5C01H
           jc          unlock_error       ;routine not shown
           write_handle stdout,u1_msg,27  ;see Function 40H
           jc          write_error        ;routine not shown
           unlock      handle,start2,lgth2 ;See Function 5C01H
           jc          unlock_error       ;routine not shown
           write_handle stdout,u2_msg,27  ;See Function 40H
           jc          write_error        ;routine not shown
           close_handle handle            ;See Function 3EH
           jc          close_error        ;routine not shown
           write_handle stdout,file,8     ;see Function 40H
           jc          write_error        ;routine not shown
           write_handle stdout,cl_msg,10  ;see Function 40H
           jc          write_error        ;routine not shown
```

Unlock (Function 5CH, Code 01H)



Call
AH = 5CH
AL = 01H
BX
  Handle
CX:DX
  Offset of area to be unlocked
SI:DI
  Length of area to be unlocked

Return
Carry set:
AX
  1 = Invalid function code
  6 = Invalid handle
  22 = Lock violation
Carry not set:
  No error

Function 5CH, Code 01H unlocks a region previously locked by the same process. BX must contain the handle of the file that contains the region to be unlocked. CX:DX (a 4-byte integer) must contain the offset in the file of the beginning of the region. SI:DI (a 4-byte integer) must contain the length of the region. The offset and length must be exactly the same as the offset and length specified in the previous Function 5CH, Code 00H (Lock).

The description of Function 5CH, Code 00H (Lock) describes how to use locked regions.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	File sharing must be loaded to use this function request.
6	The handle in BX is not a valid, open handle.
33	The region specified is not identical to one that was previously locked by the same process.

Macro Definition:

```
unlock  macro   handle,start,bytes
        mov     bx, handle
        mov     cx, word ptr start
        mov     dx, word ptr start+2
        mov     si, word ptr bytes
        mov     di, word ptr bytes+2
        mov     al, 1
        mov     ah, 5CH
        int     21H
        endm
```

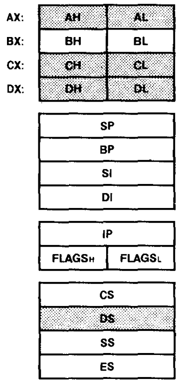
Example

The following program opens a file named FINALRPT in Deny None mode and locks two portions of it: the first 128 bytes and bytes 1024 through 5119. After some (unspecified) processing, it unlocks the same portions and closes the file.

```
stdout    equ       1
;
start1    dd        0
lgth1     dd        128
start2    dd        1023
lgth2     dd        4096
file      db       "FINALRPT",0
op_msg    db       " opened.",0DH,0AH
l1_msg    db       "First 128 bytes locked.",0DH,0AH
l2_msg    db       "Bytes 1024-5119 locked.",0DH,0AH
u1_msg    db       "First 128 bytes unlocked.",0DH,0AH
u2_msg    db       "Bytes 1024-5119 unlocked.",0DH,0AH
cl_msg    db       " closed.",0DH,0AH
handle    dw        ?
;
begin:    open_handle file,01000010b     ;see Function 3DH
          jc         open_error          ;routine not shown
          write_handle stdout,file,8     ;see Function 40H
          jc         write_error         ;routine not shown
          write_handle stdout,op_msg,10  ;see Function 40H
          jc         write_error         ;routine not shown
          mov        handle,ax           ;save handle
          lock       handle,start1,lgth1 ;See Function 5C00H
          jc         lock_error          ;routine not shown
          write_handle stdout,l1_msg,25  ;see Function 40H
          jc         write_error         ;routine not shown
          lock       handle,start2,lgth2 ;See Function 5C00H
          jc         lock_error          ;routine not shown
          write_handle stdout,l2_msg,25  ;see Function 40H
          jc         write_error         ;routine not shown
;
; ( Further processing here )
;
          unlock     handle,start1,lgth1 ;THIS FUNCTION
          jc         unlock_error        ;routine not shown
          write_handle stdout,u1_msg,27  ;see Function 40H
```

```
jc          write_error              ;routine not shown
unlock      handle,start2,lgth2 ;THIS FUNCTION
jc          unlock_error             ;routine not shown
write_handle stdout,u2_msg,27   ;see Function 40H
jc          write_error              ;routine not shown
close_handle handle                  ;See Function 3EH
jc          close_error              ;routine not shown
write_handle stdout,file,8      ;see Function 40H
jc          write_error              ;routine not shown
write_handle stdout,cl_msg,10   ;see Function 40H
jc          write_error              ;routine not shown
```

Get Machine Name (Function 5EH, Code 00H)



Call
AH = 5EH
AL = 0
DS:DX
    Pointer to 16-byte buffer

Return
Carry set:
AX
    1 = Invalid function code
Carry not set:
CX
    Identification number of local computer

Function 5EH, Code 0 retrieves the net name of the local computer. DX must contain the offset (to the segment address in DS) of a 16-byte buffer. Microsoft Networks must be running.

MS-DOS returns the local computer name (a 16-byte ASCIZ string, padded with blanks) in the buffer pointed to by DS:DX. CX returns the identification number of the local computer.

Code	Meaning
1	Microsoft Networks must be running to use this function request.

Macro Definition:

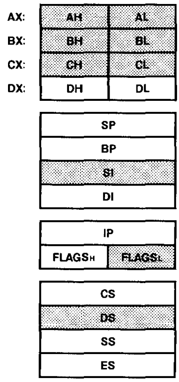
```
get_machine_name  macro  buffer
                  mov    dx,offset buffer
                  mov    al,0
                  mov    ah,5EH
                  int    21H
                  endm
```

Example

The following program displays the name of a Microsoft Networks workstation.

```
stdout equ 1
;
msg         db   "Netname: "
mac_name    db   16 dup (?),0DH,0AH
;
begin:     get_machine_name  mac_name        ;THIS FUNCTION
           jc                name_error      ;routine not shown
           write_handle      stdout,msg,27   ;see Function 40H
           jc                write_error     ;routine not shown
```

Printer Setup (Function 5EH, Code 02H)



Call
AH = 5EH
AL = 02H
BX
    Assign list index
CX
    Length of setup string

    Pointer to setup string

DS:SI
    Pointer to string

Return
Carry set:
AX
    1 = Invalid function code
Carry not set:
    No error

Function 5EH, Code 02H defines a string of control characters that MS-DOS adds to the beginning of each file sent to the network printer. BX must contain the index into the assign list that identifies the printer (entry 0 is the first entry). CX must contain the length of the string. SI must contain the offset (to the segment address in DS) of the string itself. Microsoft Networks must be running.

The setup string is added to the beginning each file sent to the printer specified by the assign list index in BX. This function request lets each program that shares a printer have its own printer configuration. You can determine which entry in the assign list refers to the printer with Function 5F02H (Get Assign List Entry).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	Microsoft Networks must be running to use this function request.

Macro Definition:

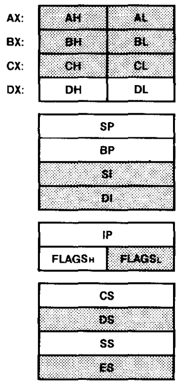
```
printer_setup  macro  index,lgth,string
               mov    bx, index
               mov    cx, lgth
               mov    dx, offset string
               mov    al, 2
               mov    ah, 5EH
               int    21H
               endm
```

Example

The following program defines a printer setup string that consists of the control character to print expanded type on Epson(R)-compatible printers. The printer cancels this mode at the first Carriage Return, so the effect is to print the first line of each file sent to the network printer as a title in expanded characters. The setup string is one character. This example assumes that the printer is the entry number 3 (the fourth entry) in the assign list. Use Function 5F02H (Get Assign List Entry) to determine this value.

```
setup     db    0EH
;
begin:    printer_setup 3,1,setup      ;THIS FUNCTION
          jc            error          ;routine not shown
```

Get Assign List Entry (Function 5FH, Code 02H)



Call
AH = 5FH
AL = 02H
BX
   Assign list index
DS:SI
   Pointer to buffer for local name
ES:DI
   Pointer to buffer for remote name

Return
Carry set:
AX
   1 = Invalid function code
   18 = No more files
Carry not set:
BL
   3 = Printer
   4 = Drive
CX
   Stored user value

Function 5FH, Code 02H retrieves the specified entry from the network list of assignments. BX must contain the assign list index (entry 0 is the first entry). SI must contain the offset (to the segment address in DS) of a 16-byte buffer for the local name. DI must contain the offset (to the segment address in ES) of a 128-byte buffer for the remote name. Microsoft Networks must be running.

MS-DOS puts the local name in the buffer pointed to by DS:SI and the remote name in the buffer pointed to by ES:DI. The local name can be a null ASCIZ string. BL returns 3 if the local device is a printer or 4 if the local device is a drive. CX returns the stored user value set with Function 5FH, Code 03H (Make Assign List Entry). The contents of the assign list can change between calls.

You can use this function request to retrieve any entry, or make a copy of the complete list by stepping through the table. To detect the end of the assign list, check for error code 18 (no more files), just as when you step through a directory with Functions 4EH and 4FH (Find First File and Find Next File).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	Microsoft Networks must be running to use this function request.
18	The index passed in BX is greater than the number of entries in the assign list.

Macro Definition:

```
get_list  macro  index,local,remote
                 mov     bx, index
                 mov     si, offset local
                 mov     di, offset remote
                 mov     al,2
                 mov     ah, 5FH
                 int     21H
                 endm
```

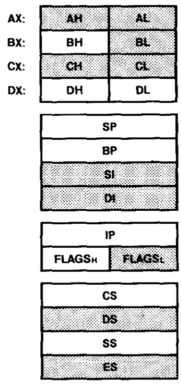
Example

The following program displays the assign list on a Microsoft Networks workstation, showing the local name, remote name, and device type (drive or printer) for each entry.

```
stdout     equ        1                    ;Code returned from
printer    equ        3                    ;GetAssignListEntry for
                                           ;a printer
header     db         13,10,13,10,"Device Type "
           db         "Local name",9 dup (20h)
           db         "Remote name"
crlf       db         13,10,13,10
header_len equ        $ - header

local_nm   db         19 dup (?)
remote_nm_len equ     $ - local_nm

remote_nm db          128 dup (?)
remote_nm_len equ     $ - remote_nm

drive_msg db          "Drive",8 dup (20h)
print_msg db          "Printer", 6 dup (20h)
device_msg_len equ    $ - print_msg

str_len    dw         ?
index      dw         ?

begin:
      write_handle stdout,header,header_len   ;see Function 40H
      jnc          set_index
      jmp          write_error
set_index:
           mov        index,0              ;assign list index
```

```
ck_list:  get_list   index,local_nm,remote_nm ;THIS FUNCTION
          jnc        got_one                  ;got an entry
          cmp        ax,18                    ;last entry?
          je         last_one                 ;yes
          jmp        return                   ;some other error
got_one:
          cmp        bl,printer               ;is it a printer?
          jc         prntr                    ;yes
          write_handle  stdout,drive_msg,device_msg_len
          jc         write_error              ;routine not shown
          jmp        short display_nms
prntr:    write_handle stdout,print_msg,device_msg_len
          jc         write_error              ;routine not shown
display_nms:
          mov        di, offset local_nm
          mov        cx,local_nm_len
          xor        ax,ax
          repne      scasb
          dec        di
          inc        cx
          mov        al,20h
          rep        stosb
          mov        di,offset remote_nm
          mov        cx,remote_nm_len
          xor        ax,ax
          repne      scasb
          dec        di
          mov        al,13
          stosb
          mov        al,10
          stosb
          mov        si,offset local_nm
          sub        di,si
          mov        str_len,di
          write_handle stdout,local_nm,str_len
          jc         write_error
          inc        index                    ;bump index
          jmp        ck_list                  ;get next entry
last_one: write_handle stdout,crlf,4          ;see Function 40H
          jc         write_error
          jmp        return
write_error:

        INCLUDE suffix.asm
```

Make Assign List Entry (Function 5FH, Code 03H)



Call
AH = 5FH
AL = 03H
BL
  3 = Printer
  4 = Drive
CX
  User value
DS:SI
  Pointer to name of source device
ES:DI
  Pointer to name of destination device

Return
Carry set:
AX
  1 = Invalid function code
  5 = Access denied
  3 = Path not found
  8 = Insufficient memory
  (Other errors particular to the network may occur.)
Carry not set:
  No error

Function 5FH, Code 03H redirects a printer or disk drive (source device) to a network directory (destination device). BL must contain 3 if the source device is a printer or 4 if the source device is a disk drive. SI must contain the offset (to the segment address in DS) of an ASCIZ string that specifies either the name of the printer, a drive letter followed by a colon, or a null string (one byte of 00H). DI must contain the offset (to the segment address in ES) of an ASCIZ string that specifies the name of a network directory. CX contains a user-specified 16-bit value that MS-DOS maintains. Microsoft Networks must be running.

The destination string must be an ASCIZ string of the following form:

<machine-name><pathname><00H><password><00H>

<machine-name> is the net name of the server that contains the network directory.

<pathname> is the alias of the network directory (not the directory path) to which the source device is to be redirected.

<00H> is a null byte.

<password> is the password for access to the network directory. If no password is specified, both null bytes must immediately follow the pathname.

If BL=3, the source string must be PRN, LPT1, LPT2, or LPT3. All output for the named printer is buffered and sent to the remote printer spooler named in the destination string.

If BL=4, the source string can be either a drive letter followed by a colon or a null string. If the source string contains a valid drive letter and colon, all subsequent references to the drive letter are redirected to the network directory named in the destination string. If the source string is a null string, MS-DOS attempts to grant access to the network directory with the specified password.

The maximum length of the destination string is 128 bytes. The value in CX can be retrieved with Function 5FH, Code 02H (Get Assign List Entry).

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	Microsoft Networks must be running to use this function request, the value in BX is not 1 to 4, the source string is in the wrong format, the destination string is in the wrong format, or the source device is already redirected.
3	The network directory path is invalid or doesn't exist.
5	The network directory/password combination is not valid. This does not mean that the password itself was invalid; the directory might not exist on the server.
8	There is not enough memory for string substitutions.

Macro Definition:

```
redir macro  device,value,source,destination
      mov    bl, device
      mov    cx, value
      mov    si, offset source
      mov    es, seg destination
      mov    di, offset destination
      mov    al, 03H
      mov    ah, 5FH
      int    21H
      endm
```

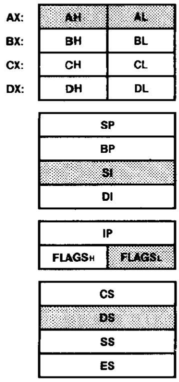
Example

The following program redirects two drives and a printer from a workstation to a server named HAROLD. It assumes the machine name, directory names, and driver letters shown:

Local drive or printer	Netname on server	Password
E:	WORD	none
F:	COMM	fred
PRN:	PRINTER	quick

```
printer     equ  3
drive       equ  4
;
local_1     db  "e:",0
local_2     db  "f:",0
local_3     db  "prn",0
remote_1    db  "\harold\word",0,0
remote_2    db  "\harold\comm",0,"fred",0
remote_3    db  "\harold\printer",0,"quick",0
;
begin:      redir  local_1,remote_1,drive,0    ;THIS FUNCTION
            jc     error                       ;routine not shown
            redir  local_2,remote_2,drive,0    ;THIS FUNCTION
            jc     error                       ;routine not shown
            redir  local_3,remote_3,printer,0 ;THIS FUNCTION
            jc     error                       ;routine not shown
```

Cancel Assign List Entry (Function 5FH, Code 04H)



Call
AH = 5FH
AL = 04H
DS:SI
   Pointer to name of source device

Return
Carry set:
AX
   1 = Invalid function code
   15 = Redirection paused on server
   (Other errors particular to the network may occur.)
Carry not set:
   No error

Function 5FH, Code 04H cancels the redirection of a printer or disk drive (source device) to a network directory (destination device) made with Function 5FH, Code 03H (Make Assign List Entry). SI must contain the offset (to the segment address in DS) of an ASCIZ string that specifies the name of the printer or drive whose redirection is to be canceled. Microsoft Networks must be running.

The ASCIZ string pointed to by DS:SI can contain one of three values:

1. The letter of a redirected drive, followed by a colon. The redirection is canceled and the drive is restored to its physical meaning.

2. The name of a redirected printer (PRN, LPT1, LPT2, or LPT3). The redirection is canceled and the printer name is restored to its physical meaning.

3. A string starting with \\ (2 backslashes). The connection between the local machine and the network directory is terminated.

If there is an error, the carry flag (CF) is set and the error code is returned in AX:

Code	Meaning
1	Microsoft Networks must be running to use this function request, or the ASCIZ string doesn't name an existing source device.
15	Disk or printer redirection on the network server is paused.

Macro Definition:

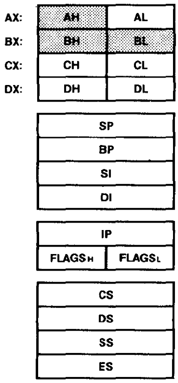
```
cancel_redir  macro   local
              mov     si, offset local
              mov     al, 4
              mov     ah, 5FH
              int     21H
              endm
```

Example

The following program cancels the redirection of drives E and F and the printer (PRN) of a Microsoft Networks workstation. It assumes that these local devices were previously redirected.

```
local_1   db           "e:",0
local_2   db           "f:",0
local_3   db           "prn",0
;
begin:    cancel_redir  local_1     ;THIS FUNCTION
          jc            error       ;routine not shown
          cancel_redir  local_2     ;THIS FUNCTION
          jc            error       ;routine not shown
          cancel_redir  local_3     ;THIS FUNCTION
          jc            error       ;routine not shown
```

Get PSP (Function 62H)



Call
AH = 62H

Return
BX
Segment address of the Program Segment Prefix of the current process

Function 62H retrieves the segment address of the currently active process (the start of the Program Segment Prefix). The address is returned in BX.

Macro Definition:

```
get_psp macro
        mov     ah, 62H
        int     21H
        endm
```

Example

The following program displays the segment address of its Program Segment Prefix (PSP) in hexadecimal.

```
msg       db        "PSP segment address:  H",0DH,0AH,"$"
;
begin:    get_psp                              ;THIS FUNCTION
          convert   bx,16,msg[21]              ;see end of chapter
          display   msg                        ;see Function 09H
```

```
; MACRO DEFINITIONS FOR MS-DOS SYSTEM CALL EXAMPLES
;
;******************
; Interrupts
;******************
;                                    INTERRUPT 25H
ABS_DISK_READ  macro  disk,buffer,num_sectors,first_sector
     mov    al,disk
     mov    bx,offset buffer
     mov    cx,num_sectors
     mov    dx,first_sector
     int    25H
     popf
     endm
;                                    INTERRUPT 26H
ABS_DISK_WRITE  macro  disk,buffer,num_sectors,first_sector
     mov    al,disk
     mov    bx,offset buffer
     mov    cx,num_sectors
     mov    dx,first_sector
     int    26H
     popf
     endm
;                                    INTERRUPT 27H
STAY_RESIDENT  macro  last_instruc
     mov    dx,offset last_instruc
     inc    dx
     int    27H
     endm
;
;
;******************
; Function Requests
;******************
;                                    FUNCTION REQUEST 00H
TERMINATE_PROGRAM  macro
     xor    ah,ah
     int    21H
     endm
;                                    FUNCTION REQUEST 01H
READ_KBD_AND_ECHO  macro
     mov    ah,01H
     int    21H
     endm
;                                    FUNCTION REQUEST 02H
DISPLAY_CHAR  macro  character
     mov    dl,character
     mov    ah,02H
     int    21H
     endm
;                                    FUNCTION REQUEST 03H
AUX_INPUT    macro
     mov     ah,03H
     int     21H
     endm
```

```
;                                      FUNCTION REQUEST 04H
AUX_OUTPUT  macro
     mov    ah,04H
     int    21H
     endm
;                                      FUNCTION REQUEST 05H
PRINT_CHAR  macro  character
     mov    dl,character
     mov    ah,05H
     int    21H
     endm
;                                      FUNCTION REQUEST 06H
DIR_CONSOLE_IO  macro  switch
     mov    dl,switch
     mov    ah,06H
     int    21H
     endm
;                                      FUNCTION REQUEST 07H
DIR_CONSOLE_INPUT  macro
     mov    ah,07H
     int    21H
     endm
;                                      FUNCTION REQUEST 08H
READ_KBD  macro
     mov    ah,08H
     int    21H
     endm
;                                      FUNCTION REQUEST 09H
DISPLAY  macro  string
     mov    dx,offset string
     mov    ah,09H
     int    21H
     endm
;                                      FUNCTION REQUEST 0AH
GET_STRING  macro  limit,string
     mov    dx,offset string
     mov    string,limit
     mov    ah,0AH
     int    21H
     endm
;                                      FUNCTION REQUEST 0BH
CHECK_KBD_STATUS  macro
     mov    ah,0BH
     int    21H
     endm
;                                      FUNCTION REQUEST 0CH
FLUSH_AND_READ_KBD  macro  switch
     mov    al,switch
     mov    ah,0CH
     int    21H
     endm
```

```
;                                       FUNCTION REQUEST 0DH
RESET_DISK  macro
     mov    ah,0DH
     int    21H
     endm
;                                       FUNCTION REQUEST 0EH
SELECT_DISK  macro  disk
     mov    dl,disk[-65]
     mov    ah,0EH
     int    21H
     endm
;                                       FUNCTION REQUEST 0FH
OPEN  macro  fcb
     mov    dx,offset fcb
     mov    ah,0FH
     int    21H
     endm
;                                       FUNCTION REQUEST 10H
CLOSE  macro  fcb
     mov    dx,offset fcb
     mov    ah,10H
     int    21H
     endm
;                                       FUNCTION REQUEST 11H
SEARCH_FIRST  macro  fcb
     mov    dx,offset fcb
     mov    ah,11H
     int    21H
     endm
;                                       FUNCTION REQUEST 12H
SEARCH_NEXT  macro  fcb
     mov    dx,offset fcb
     mov    ah,12H
     int    21H
     endm
;                                       FUNCTION REQUEST 13H
DELETE  macro  fcb
     mov    dx,offset fcb
     mov    ah,13H
     int    21H
     endm
;                                       FUNCTION REQUEST 14H
READ_SEQ  macro  fcb
     mov    dx,offset fcb
     mov    ah,14H
     int    21H
     endm
;                                       FUNCTION REQUEST 15H
WRITE_SEQ  macro  fcb
     mov    dx,offset fcb
     mov    ah,15H
     int    21H
     endm
```

```
;                                FUNCTION REQUEST 16H
CREATE  macro  fcb
     mov     dx,offset fcb
     mov     ah,16H
     int     21H
     endm
;                                FUNCTION REQUEST 17H
RENAME  macro  fcb,newname
     mov     dx,offset fcb
     mov     ah,17H
     int     21H
     endm
;                                FUNCTION REQUEST 19H
CURRENT_DISK  macro
     mov     ah,19H
     int     21H
     endm
;                                FUNCTION REQUEST 1AH
SET_DTA  macro  buffer
     mov     dx,offset buffer
     mov     ah,1AH
     endm
;                                FUNCTION REQUEST 1BH
DEF_DRIVE_DATA  macro
     mov     ah,1BH
     int     21H
     endm
;                                FUNCTION REQUEST 1CH
DRIVE_DATA  macro  drive
     mov     dl,drive
     mov     ah,1CH
     int     21H
     endm
;                                FUNCTION REQUEST 21H
READ_RAN  macro  fcb
     mov     dx,offset fcb
     mov     ah,21H
     int     21H
     endm
;                                FUNCTION REQUEST 22H
WRITE_RAN  macro  fcb
     mov     dx,offset fcb
     mov     ah,22H
     int     21H
     endm
;                                FUNCTION REQUEST 23H
FILE_SIZE  macro  fcb
     mov     dx,offset fcb
     mov     ah,23H
     int     21H
     endm
```

```
;                                         FUNCTION REQUEST 24H
SET_RELATIVE_RECORD  macro   fcb
     mov     dx,offset fcb
     mov     ah,24H
     int     21H
     endm
;                                         FUNCTION REQUEST 25H
SET_VECTOR  macro  interrupt,handler_start
     mov     al,interrupt
     mov     dx,offset handler_start
     mov     ah,25H
     int     21H
     endm
;                                         FUNCTION REQUEST 26H
CREATE_PSP  macro  seg_addr
     mov     dx,offset seg_addr
     mov     ah,26H
     int     21H
     endm
;                                          FUNCTION REQUEST 27H
RAN_BLOCK_READ  macro  fcb,count,rec_size
     mov     dx,offset fcb
     mov     cx,count
     mov     word ptr fcb[14],rec_size
     mov     ah,27H
     int     21H
     endm
;                                          FUNCTION REQUEST 28H
RAN_BLOCK_WRITE  macro  fcb,count,rec_size
     mov     dx,offset fcb
     mov     cx,count
     mov     word ptr fcb[14],rec_size
     mov     ah,28H
     int     21H
     endm
;                                         FUNCTION REQUEST 29H
PARSE  macro  string,fcb
     mov     si,offset string
     mov     di,offset fcb
     push    es
     push    ds
     pop     es
     mov     al,0FH
     mov     ah,29H
     int     21H
     pop     es
     endm
;                                          FUNCTION REQUEST 2AH
GET_DATE  macro
     mov     ah,2AH
     int     21H
     endm
```

```
;                                    FUNCTION REQUEST 2BH
SET_DATE  macro  year,month,day
     mov    cx,year
     mov    dh,month
     mov    dl,day
     mov    ah,2BH
     int    21H
     endm
;                                    FUNCTION REQUEST 2CH
GET_TIME  macro
     mov    ah,2CH
     int    21H
     endm
;                                    FUNCTION REQUEST 2DH
SET_TIME  macro  hour,minutes,seconds,hundredths
     mov    ch,hour
     mov    cl,minutes
     mov    dh,seconds
     mov    dl,hundredths
     mov    ah,2DH
     int    21H
     endm
;                                    FUNCTION REQUEST 2EH
VERIFY  macro  switch
     mov    al,switch
     mov    ah,2EH
     int    21H
     endm
;                                    FUNCTION REQUEST 2FH
GET_DTA  macro
     mov    ah,2FH
     int    21H
     endm
;                                    FUNCTION REQUEST 30H
GET_VERSION  macro
     mov    ah,30H
     int    21H
     endm
;                                    FUNCTION REQUEST 31H
KEEP_PROCESS  macro  return_code,last_byte
     mov    al,return_code
     mov    dx,offset last_byte
     mov    cl,4
     shr    dx,cl
     inc    dx
     mov    ah,31H
     int    21H
     endm
;                                    FUNCTION REQUEST 33H
CTRL_C_CK  macro  action,state
     mov    al,action
     mov    dl,state
     mov    ah,33H
     int    21H
     endm
```

```
;                                        FUNCTION REQUEST 35H
GET_VECTOR  macro   interrupt
     mov    al,interrupt
     mov    ah,35H
     int    21H
     endm
;                                        FUNCTION REQUEST 36H
GET_DISK_SPACE  macro  drive
     mov    dl,drive
     mov    ah,36H
     int    21H
     endm
;                                        FUNCTION REQUEST 38H
GET_COUNTRY   macro   country,buffer
              local   gc_01
              mov     dx,offset buffer
              mov     ax,country
              cmp     ax,0FFH
              jl      gc_01
              mov     al,0ffh
              mov     bx,country
gc_01:        mov     ah,38H
              int     21H
              endm
;                                        FUNCTION REQUEST 38H
SET_COUNTRY   macro   country
              local   sc_01
              mov     dx,0FFFFH
              mov     ax,country
              cmp     ax,0FFH
              jl      sc_01
              mov     al,0ffh
              mov     bx,country
sc_01:        mov     ah,38H
              int     21H
              endm
;                                        FUNCTION REQUEST 39H
MAKE_DIR  macro  path
     mov    dx,offset path
     mov    ah,39H
     int    21H
     endm
;                                        FUNCTION REQUEST 3AH
REM_DIR  macro  path
     mov    dx,offset path
     mov    ah,3AH
     int    21H
     endm
;                                        FUNCTION REQUEST 3BH
CHANGE_DIR  macro  path
     mov    dx,offset path
     mov    ah,3BH
     int    21H
     endm
```

```
;                                  FUNCTION REQUEST 3CH
CREATE_HANDLE  macro  path,attrib
     mov      dx,offset path
     mov      cx,attrib
     mov      ah,3CH
     int      21H
     endm
;                                  FUNCTION REQUEST 3DH
OPEN_HANDLE  macro  path,access
     mov      dx,offset path
     mov      al,access
     mov       ah,3DH
     int      21H
     endm
;                                  FUNCTION REQUEST 3EH
CLOSE_HANDLE  macro  handle
     mov      bx,handle
     mov      ah,3EH
     int      21H
     endm
;                                  FUNCTION REQUEST 3FH
READ_HANDLE  macro  handle,buffer,bytes
     mov      bx,handle
     mov      dx,offset buffer
     mov      cx,bytes
     mov      ah,3FH
     int      21H
     endm
;                                  FUNCTION REQUEST 40H
WRITE_HANDLE  macro  handle,buffer,bytes
     mov      bx,handle
     mov      dx,offset buffer
     mov      cx,bytes
     mov      ah,40H
     int      21H
     endm
;                                  FUNCTION REQUEST 41H
DELETE_ENTRY  macro  path
     mov      dx,offset path
     mov      ah,41H
     int      21H
     endm
;                                  FUNCTION REQUEST 42H
MOVE_PTR  macro  handle,high,low,method
     mov      bx,handle
     mov      cx,high
     mov      dx,low
     mov      al,method
     mov      ah,42H
     int      21H
     endm
```

```
;                                   FUNCTION REQUEST 43H
CHANGE_MODE  macro  path,action,attrib
     mov    dx,offset path
     mov    al,action
     mov    cx,attrib
     mov    ah,43H
     int    21H
     endm
;                                   FUNCTION REQUEST 4400H,01H
IOCTL_DATA  macro  code,handle
     mov    bx,handle
     mov    al,code
     mov    ah,44H
     int    21H
     endm
;                                   FUNCTION REQUEST 4402H,03H
IOCTL_CHAR  macro  code,handle,buffer
     mov    bx,handle
     mov    dx,offset buffer
     mov    al,code
     mov    ah,44H
     int    21H
     endm
;                                   FUNCTION REQUEST 4404H,05H
IOCTL_STATUS  macro  code,drive,buffer
     mov    bl,drive
     mov    dx,offset buffer
     mov    al,code
     mov    ah,44H
     int    21H
     endm
;                                   FUNCTION REQUEST 4406H,07H
IOCTL_BLOCK  macro  code,handle
     mov    bx,handle
     mov    al,code
     mov    ah,44H
     int    21H
     endm
;                                   FUNCTION REQUEST 4408H
IOCTL_CHANGE  macro  drive
     mov    bl,drive
     mov    al,08H
     mov    ah,44H
     int    21H
     endm
;                                   FUNCTION REQUEST 4409H
IOCTL_RBLOCK  macro  drive
     mov    bl,drive
     mov    al,09H
     mov    ah,44H
     int    21H
     endm
```

```
;                                         FUNCTION REQUEST 440AH
IOCTL_RHANDLE  macro  handle
     mov      bx,handle
     mov      al,0AH
     mov      ah,44H
     int      21H
     endm
;                                         FUNCTION REQUEST 440BH
IOCTL_RETRY  macro  retries,wait
     mov      bx,retries
     mov      cx,wait
     mov      al,0BH
     mov      ah,44H
     int      21H
     endm
;                                         FUNCTION REQUEST 45H
XDUP  macro  handle
     mov      bx,handle
     mov      ah,45H
     int      21H
     endm
;                                         FUNCTION REQUEST 46H
XDUP2  macro  handle1,handle2
     mov      bx,handle1
     mov      cx,handle2
     mov      ah,46H
     int      21H
     endm
;                                         FUNCTION REQUEST 47H
GET_DIR  macro  drive,buffer
     mov      dl,drive
     mov      si,offset buffer
     mov      ah,47H
     int      21H
     endm
;                                         FUNCTION REQUEST 48H
ALLOCATE_MEMORY  macro  bytes
     mov      bx,bytes
     mov      cl,4
     shr            bx,cl
     inc      bx
     mov      ah,48H
     int      21H
     endm
;                                         FUNCTION REQUEST 49H
FREE_MEMORY  macro  seg_addr
     mov      ax,seg_addr
     mov      es,ax
     mov      ah,49H
     int      21H
     endm
```

```
;                                        FUNCTION REQUEST 4AH
SET_BLOCK  macro  last_byte
     mov    bx,offset last_byte
     mov    cl,4
     shr    bx,cl
     add    bx,17
     mov    ah,4AH
     int    21H
     mov     ax,bx
     shl     ax,cl
     mov     sp,ax
     mov     bp,sp
     endm
;                                        FUNCTION REQUEST 4B00H
EXEC macro  path,command,parms
     mov    dx,offset path
     mov    bx,offset parms
     mov    word ptr parms[02h],offset command
     mov    word ptr parms[04h],cs
     mov    word ptr parms[06h],5ch
     mov    word ptr parms[08h],es
     mov    word ptr parms[0ah],6ch
     mov    word ptr parms[0ch],es
     mov    al,0
     mov    ah,4BH
     int    21H
     endm
;                                        FUNCTION REQUEST 4B03H
EXEC_OVL  macro  path,parms,seg_addr
      mov    dx,offset path
      mov    bx,offset parms
      mov    parms,seg_addr
      mov    parms[02H],seg_addr
      mov    al,3
      mov    ah,4BH
      int    21H
      endm
;                                        FUNCTION REQUEST 4CH
END_PROCESS  macro  return_code
             mov    al,return_code
             mov    ah,4CH
             int    21H
             endm
;                                        FUNCTION REQUEST 4DH
WAIT  macro
      mov    ah,4DH
      int    21H
      endm
;                                        FUNCTION REQUEST 4EH
FIND_FIRST_FILE  macro  path,attrib
     mov    dx,offset path
     mov    cx,attrib
     mov    ah,4EH
     int    21H
     endm
```

```
;                                       FUNCTION REQUEST 4FH
FIND_NEXT_FILE  macro
     mov     ah,4FH
     int     21H
     endm
;                                       FUNCTION REQUEST 54H
GET_VERIFY   macro
     mov     ah,54H
     int     21H
     endm
;                                       FUNCTION REQUEST 56H
RENAME_FILE   macro   old_path,new_path
     mov     dx,offset old_path
     push    ds
     pop     es
     mov     di,offset new_path
     mov     ah,56H
     int     21H
     endm
;                                       FUNCTION REQUEST 57H
GET_SET_DATE_TIME   macro   handle,action,time,date
     mov      bx,handle
     mov     al,action
     mov     cx,word ptr time
     mov     dx,word ptr date
     mov     ah,57H
     int     21H
     endm
;                                       FUNCTION REQUEST 58H
ALLOC_STRAT   macro   code,strategy
     mov     bx,strategy
     mov     al,code
     mov     ah,58H
     int     21H
     endm
;                                       FUNCTION REQUEST 59H
GET_ERROR   macro
     mov     ah,59
     int     21H
     endm
;                                       FUNCTION REQUEST 5AH
CREATE_TEMP   macro   pathname,attrib
              mov     cx,attrib
              mov     dx,offset pathname
              mov     ah,5AH
              int     21H
              endm
;                                       FUNCTION REQUEST 5BH
CREATE_NEW   macro   pathname,attrib
             mov     cx,attrib
             mov     dx,offset pathname
             mov     ah,5BH
             int     21H
             endm
```

```
;                                       FUNCTION REQUEST 5C00H
LOCK            macro   handle,start,bytes
                mov     bx,handle
                mov     cx,word ptr start
                mov     dx,word ptr start+2
                mov     si,word ptr bytes
                mov     di,word ptr bytes+2
                mov     al,0
                mov     ah,5CH
                int     21H
                endm
;                                       FUNCTION REQUEST 5C01H
UNLOCK          macro   handle,start,bytes
                mov     bx,handle
                mov     cx,word ptr start
                mov     dx,word ptr start+2
                mov     si,word ptr bytes
                mov     di,word ptr bytes+2
                mov     al,1
                mov     ah,5CH
                int     21H
                endm
;                                       FUNCTION REQUEST 5E00H
GET_MACHINE_NAME  macro  buffer
                mov     dx,offset buffer
                mov     al,0
                mov     ah,5EH
                int     21H
                endm
;                                       FUNCTION REQUEST 5E02H
PRINTER_SETUP  macro  index,lgth,string
                mov     bx,index
                mov     cx,lgth
                mov     dx,offset string
                mov     al,2
                mov     ah,5EH
                int     21H
                endm
;                                       FUNCTION REQUEST 5F02H
GET_LIST        macro   index,local,remote
                mov     bx,index
                mov     si,offset local
                mov     di,offset remote
                mov     al,2
                mov     ah,5FH
                int     21H
                endm
```

```
;                                       FUNCTION REQUEST 5F03H
REDIR           macro  local,remote,device,value
                mov    bl,device
                mov    cx,value
                mov    si,offset local
                mov    di,offset remote
                mov    al,3
                mov    ah,5FH
                int    21H
                endm
;                                       FUNCTION REQUEST 5F04H
CANCEL_REDIR  macro  local
                mov    si,offset local
                mov    al,4
                mov    ah,5FH
                int    21H
                endm
;                                       FUNCTION REQUEST 62H
GET_PSP         macro
                mov    ah,62H
                int    21H
                endm
;
;
;*******************
; General
;*******************
;
DISPLAY_ASCIIZ  macro  asciiz_string
     local   search,found_it
     mov     bx,offset asciiz_string

search:
     cmp     byte ptr [bx],0
     je      found_it
     inc     bx
     jmp short search

found_it:
     mov     byte ptr [bx],"$"
     display asciiz_string
     mov     byte ptr [bx],0
     display_char 0DH
     display_char 0AH
     endm
```

```
;
MOVE_STRING  macro  source,destination,count
     push   es
     push   ds
     pop    es
     assume es:code
     mov    si,offset source
     mov    di,offset destination
     mov    cx,count
 rep movs   es:destination,source
     assume es:nothing
     pop    es
     endm
;
CONVERT  macro  value,base,destination
     local  table,start
     jmp    start
table  db   "0123456789ABCDEF"

start:
     push   ax
     push   bx
     push   dx
     mov    al,value
     xor    ah,ah
     xor    bx,bx
     div    base
     mov    bl,al
     mov    al,cs:table[bx]
     mov    destination,al
     mov    bl,ah
     mov    al,cs:table[bx]
     mov    destination[1],al
     pop    dx
     pop    bx
     pop    ax
     endm
;
CONVERT_TO_BINARY  macro  string,number,value
     local  ten,start,calc,mult,no_mult
     jmp    start
ten  db     10

start:
     mov    value,0
     xor    cx,cx
     mov    cl,number
     xor    si,si
```

```
calc:
     xor     ax,ax
     mov     al,string[si]
     sub     al,48
     cmp     cx,2
     jl      no_mult
     push    cx
     dec     cx

mult:
     mul     cs:ten
     loop    mult
     pop     cx

no_mult:
     add     value,ax
     inc     si
     loop    calc
     endm
;
CONVERT_DATE  macro  dir_entry
     mov     dx,word ptr dir_entry[24]
     mov     cl,5
     shr     dl,cl
     mov     dh,dir_entry[24]
     and     dh,1FH
     xor     cx,cx
     mov     cl,dir_entry[25]
     shr     cl,1
     add     cx,1980
     endm
;
PACK_DATE  macro  date
     local set_bit
;
; On entry: DH=day, DL=month, CX=(year-1980)
;
     sub    cx,1980
     push   cx
     mov    date,dh
     mov    cl,5
     shl    dl,cl
     pop    cx
     jnc    set_bit
     or     cl,80h

set_bit:
     or     date,dl
     rol    cl,1
     mov    date[1],cl
     endm
;
```

Chapter 2
MS-DOS Device Drivers

CHAPTER 2

MS-DOS DEVICE DRIVERS

2.1 INTRODUCTION

The IO.SYS file is composed of the "resident" device drivers. This forms the MS-DOS BIOS, and these drivers are called upon by MS-DOS to handle I/O requests initiated by application programs.

One of the most powerful features of MS-DOS is the ability to add new devices such as printers, plotters, or mouse input devices without rewriting the BIOS. The MS-DOS BIOS is "configurable;" that is, new drivers can be added and existing drivers can be pre-empted. Non-resident device drivers may be easily added by an end user at boot time via the "DEVICE =" entry in the CONFIG.SYS file. In this section, these non-resident drivers are termed "installable" to distinguish them from drivers in the IO.SYS file, which are considered the resident drivers.

At boot time, a minimum of five resident device drivers must be present. These drivers are in a linked list: the "header" of each one contains a DWORD pointer to the next. The last driver in the chain has an end-of-list marker of -1, -1 (all bits on).

Each driver in the chain has two entry points: the strategy entry point and the interrupt entry point. MS-DOS does not take advantage of the two entry points: it calls the strategy routine, then immediately calls the interrupt routine.

The dual entry points facilitate future multitasking versions of MS-DOS. In multitasking environments, I/O must be asynchronous; to accomplish this, the strategy routine will be called to (internally) queue a request and return quickly. It is then the responsibility of the interrupt routine to perform the I/O at interrupt time by getting requests from the internal queue and processing them. When a request is completed, it is flagged as "done" by the interrupt routine. MS-DOS periodically scans the list of requests looking for those that are flagged as done, and "wakes up" the process waiting for the completion of the request.

When requests are queued in this manner, it is no longer sufficient to pass I/O information in registers, since many requests may be pending at any time. Therefore, the MS-DOS device interface uses "packets" to pass request information. These request packets are of variable size and format, and are composed of two parts:

1. The static request header section, which has the same format for all requests.

2. A section which has information specific to the type of request.

A driver is called with a pointer to a packet. In multitasking versions, this packet will be linked into a global chain of all pending I/O requests maintained by MS-DOS.

MS-DOS does not implement a global or local queue. Only one request is pending at any one time. The strategy routine must store the address of the packet at a fixed location, and the interrupt routine, which is called immediately after the strategy routine, should process the packet by completing the request and returning. It is assumed that the request is completed when the interrupt routine returns.

To make a device driver that SYSINIT can install, a .BIN (core image) or .EXE format file must be created with the device driver header at the beginning of the file. The link field should be initialized to -1 (SYSINIT fills it in). Device drivers which are part of the BIOS should have their headers point to the next device in the list and the last header should be initialized to -1,-1. The BIOS must be a .BIN (core image) format file.

.EXE format installable device drivers may be used in non-IBM versions of MS-DOS. On the IBM PC, the .EXE loader is located in COMMAND.COM which is not present at the time that installable devices are being loaded.

2.2 FORMAT OF A DEVICE DRIVER

A device driver is a program segment responsible for communication between DOS and the system hardware. It has a special header at the beginning identifying it as a device driver, defining entry points, and describing various attributes of the device.

Note

For device drivers, the file must not use the ORG 100H (like .COM files). Because it does not use the Program Segment Prefix, the device driver is simply loaded; therefore, the file must have an origin of zero (ORG 0 or no ORG statement).

There are two kinds of device drivers:

1. Character device drivers
2. Block device drivers

Character devices perform serial character I/O. Examples are the console, communications port and printer. These devices are named (i.e., CON, AUX, CLOCK, etc.), and programs may open channels (handles or FCBs) to do I/O to them.

Block devices are the "disk drives" on the system. They can perform random I/O in structured pieces called blocks (usually the physical sector size). These devices are not named as the character devices are, and therefore cannot be opened directly. Instead they have unit numbers and are identified by driver letters such as A, B, and C.

A single block-device driver may be responsible for one or more logically contiguous disk drives. For example, block device driver ALPHA may be responsible for drives A, B, C, and D. This means that it has four units defined (0-3), and therefore, takes up four drive letters. The position of the driver in the list of all drivers determines which units correspond to which driver letters. If driver ALPHA is the first block driver in the device list, and it defines 4 units (0-3), then they will be A, B, C, and D. If BETA is the second block driver and defines three units (0-2), then they will be E, F, and G, and so on. The theoretical limit is 63, but it should be noted that the device installation code will not allow the installation of a device if it would result in a drive letter >'Z' (5AH). All block device drivers present in the standard resident BIOS will be placed ahead of installable block-device drivers in the list.

Note

Character devices cannot define multiple units because they have only one name.

2.3 HOW TO CREATE A DEVICE DRIVER

To create a device driver that MS-DOS can install, you must create a binary file (.COM or .EXE format) with a device header at the beginning of the file. Note that for device drivers, the code should not be originated at 100H, but at 0. The device header contains a link field (pointer to next device header) which should be -1, unless there is more than one device driver in the file. The attribute field and entry points must be set correctly.

If it is a character device, the name field should be filled in with the name of that character device. The name can be any legal 8-character filename. If the name is less than eight characters, it should be padded out to eight characters with spaces (20H). Note that device names do not include colons (:). The fact that "CON" is the same as "CON:" is a property of the default MS-DOS command interpreter (COMMAND.COM) and not the device driver or the MS-DOS interface. All character device names are handled in this way.

MS-DOS always processes installable device drivers before handling the default devices, so to install a new CON device, simply name the device "CON". Remember to set the standard input device and standard output device bits in the attribute word on a new CON device. The scan of the device list stops on the first match, so the installable device driver takes precedence.

It is not possible to replace the "resident" disk block device driver with an installable device driver the same way you can replace the other device drivers in the BIOS. Block drivers can be used only for devices not directly supported by the default disk drivers in IO.SYS.

> **Note**
>
> Because MS-DOS can install the driver anywhere in memory, care must be taken when making far memory references. You should not expect that your driver will always be loaded in the same place every time.

2.3.1 Device Strategy Routine

This routine, which is called by MS-DOS for each device driver service request, is primarily responsible for queuing these requests in the order in which they are to be processed by the Device Interrupt Routine. Such queuing can be a very important performance feature in a multitasking environment, or where asynchronous I/O is supported. As MS-DOS does not currently support these facilities, only one request can be serviced at a time, and this routine is usually very short. In the coding examples in Section 2.12, each request is simply stored in a single pointer area.

2.3.2 Device Interrupt Routine

This routine contains all of the code to process the service request. It may actually interface to the hardware, or it may use ROM BIOS calls. It usually consists of a series of procedures which handle the specific command codes to be supported as well as some exit and error-handling routines. See the coding examples in Section 2.12.

2.4 INSTALLATION OF DEVICE DRIVERS

MS-DOS allows new device drivers to be installed dynamically at boot time. This is accomplished by initialization code in IO.SYS which reads and processes the CONFIG.SYS file.

MS-DOS calls upon the device drivers to perform their function in the following manner:

1. MS-DOS makes a far call to strategy entry.

2. MS-DOS passes device driver information in a request header to the strategy routine.

3. MS-DOS makes a far call to the interrupt entry.

This structure is designed to be easily upgraded to support any future multitasking environment.

2.5 DEVICE HEADERS

A device header is required at the beginning of a device driver. A device header looks like this:

```
+---------------------------------------------+
| DWORD Pointer to next device                |
| (Usually set to -1 if this driver           |
| is the last or only driver in the           |
| file)                                       |
+---------------------------------------------+
| WORD Attributes                             |
|  Bit 15 = 1 if character device             |
|         = 0 if block device                 |
|  Bit 14 = 1 if IOCTL supported              |
|  Bit 13 = 1 if output till busy             |
|               (character devices)           |
|         = 1 if NON FAT ID                   |
|               (block devices)               |
|  Bit 12 = reserved (must be 0)              |
|  Bit 11 = 1 if support OPEN/CLOSE/RM        |
|  Bit 10-5 reserved (must be 0)              |
|  Bit 3 = 1 if intended current CLOCK        |
|              device                         |
|  Bit 2 = 1 if intended current NUL          |
|              device                         |
|  Bit 1 = 1 if intended current sto          |
|              device                         |
|  Bit 0 = 1 if intended current sti          |
|              device                         |
+---------------------------------------------+
| WORD Pointer to device strategy             |
|      entry point                            |
+---------------------------------------------+
| WORD Pointer to device interrupt            |
|      entry point                            |
+---------------------------------------------+
| 8-BYTE Character device name field          |
| Character devices set a device name.        |
| For block devices the first byte is         |
| the number of units.                        |
+---------------------------------------------+
```

Figure 2.1 Sample Device Header

Note that the device entry points are words. They must be offsets from the same segment number used to point to this table. For example, if XXX:YYY points to the start of this table, then XXX:strategy and XXX:interrupt are the entry points.

The device header fields are described in the following section.

2.5.1 Pointer to Next Device Field

The pointer to the next device header field is a double word field (offset followed by segment) that is set by MS-DOS to point at the next driver in the system list at the time the device driver is loaded. It is important that this field be set to -1 prior to load (when it is on the disk as a file) unless there is more than one device driver in the file. If there is more than one driver in the file, the first word of the double word pointer should be the offset of the next driver's device header.

Note

If there is more than one device driver in the file, the last driver in the file must have the pointer to the next device header field set to -1.

2.5.2 Attribute Field

The attribute field is used to identify the type of device this driver is responsible for. In addition to distinguishing between block and character devices, these bits are used to give selected character devices special treatment. (Note that if a bit in the attribute word is defined only for one type of device, a driver for the other type of device must set that bit to 0.)

For example, assume that a user has a new device driver that he wants to use as the standard input and output. In addition to installing the driver, he must tell MS-DOS that he wants his new driver to override the current standard input and standard output (the CON device). This is accomplished by setting the attributes to the desired characteristics, so he would set bits 0 and 1 to 1 (note that they are separate!). Similarly, a new CLOCK device could be installed by setting that attribute. (Refer to Section 2.10, "The CLOCK Device," in this chapter for more information.) Although there is a NUL device attribute, the NUL device cannot be reassigned. This attribute exists so that MS-DOS can determine if the NUL device is being used.

The NON FAT ID bit for block devices affects the operation of the BUILD BPB (BIOS Parameter Block) device call. The NON FAT ID bit has a different meaning on character devices. It indicates that the device implements the OUTPUT UNTIL BUSY device call.

The IOCTL bit has meaning on character and block devices.

The IOCTL functions allow data to be sent and received by the device for its own use (for example, to set baud rate, stop bits, and form length), instead of passing data over the device channel as does a normal read or write. The interpretation of the passed information is up to the device, but it must not be treated as a normal I/O request. This bit tells MS-DOS whether the device can handle control strings via the IOCTL system call, Function 44H.

If a driver cannot process control strings, it should initially set this bit to 0. This tells MS-DOS to return an error if an attempt is made (via Function 44H) to send or receive control strings to this device. A device which can process control strings should initialize the IOCTL bit to 1. For drivers of this type, MS-DOS will make calls to the IOCTL INPUT and OUTPUT device functions to send and receive IOCTL strings.

The OPEN/CLOSE/RM bit signals to MS-DOS 3.x and later versions whether this driver supports additional MS-DOS 3.0 functionality. To support these old drivers, it is necessary to detect them. This bit was reserved in MS-DOS 2.x, and is 0. All new devices should support the OPEN, CLOSE, and REMOVABLE MEDIA calls and set this bit to 1. Since MS-DOS 2.x never makes these calls, the driver will be backward compatible.

2.5.3 Strategy And Interrupt Routines

These two fields are the pointers to the entry points of the strategy and interrupt routines. They are word values, so they must be in the same segment as the device header.

2.5.4 Name Field

This is an 8-byte field that contains the name of a character device or the number of units of a block device. If it is a block device, the number of units can be put in the first byte. This is optional, because MS-DOS will fill in this location with the value returned by the driver's INIT code. Refer to Section 2.4, "Installation of Device Drivers," for more information.

2.6 REQUEST HEADER

When MS-DOS calls a device driver to perform a function, it passes a request header in ES:BX to the strategy entry point. This is a fixed length header, followed by data pertinent to the operation being performed. Note that it is the device driver's responsibility to preserve the machine state (for example, save all registers including flags on entry and restore them on exit). There is enough room on the stack when strategy or interrupt is called to do about 20 pushes. If more stack is needed, the driver should set up its own stack.

The following figure illustrates a request header.

REQUEST HEADER ->

BYTE Length of record Length in bytes of this request header
BYTE Unit code The subunit the operation is for (minor device) (no meaning on character devices)
BYTE Command code
WORD Status
8 BYTES Reserved

Figure 2.2 Request Header

The request header fields are described below.

2.6.1 Length of Record

This field contains the length (in bytes) of the request header.

2.6.2 Unit Code Field

The unit code field identifies which unit in your device driver the request is for. For example, if your device driver has 3 units defined, then the possible values of the unit code field would be 0, 1, and 2.

2.6.3 Command Code Field

The command code field in the request header can have the following values:

Command Code	Function
0	INIT
1	MEDIA CHECK (Block devices only)
2	BUILD BPB " " "
3	IOCTL INPUT (Only called if device has IOCTL)
4	INPUT (read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Char devs only)
6	INPUT STATUS " " "
7	INPUT FLUSH " " "
8	OUTPUT (Write)
9	OUTPUT (Write) with verify
10	OUTPUT STATUS " " "
11	OUTPUT FLUSH " " "
12	IOCTL OUTPUT (Only called if device has IOCTL)
13	DEVICE OPEN (Only called if OPEN/CLOSE/RM bit set)
14	DEVICE CLOSE (Only called if OPEN/CLOSE/RM bit set)
15	REMOVABLE MEDIA (Only called if OPEN/CLOSE/RM bit set and device is block)
16	OUTPUT UNTIL BUSY (Only called if bit 13 is set on character devices)

2.6.4 Status Field

The following figure illustrates the status field in the request header.

15	14 13 12 11 10	9	8	7 6 5 4 3 2 1 0
E R R	RESERVED	B U S Y	D O N E	ERROR CODE (bit 15 on)

The status word is zero on entry and is set by the driver interrupt routine on return.

Bit 8 is the done bit. When set, it means the operation has completed. The driver sets it to 1 when it exits.

Bit 15 is the error bit. If it is set, then the low 8 bits indicate the error. The errors are:

0 Write protect violation
1 Unknown unit
2 Drive not ready
3 Unknown command
4 CRC error
5 Bad drive request structure length
6 Seek error
7 Unknown media
8 Sector not found
9 Printer out of paper
A Write fault
B Read fault
C General failure
D Reserved
E Reserved
F Invalid disk change

Bit 9 is the busy bit, which is set only by status calls and the removable media call.

2.7 DEVICE DRIVER FUNCTIONS

Device drivers may perform all or some of these nine general functions. In some cases, these functions break down into several command codes, for specific cases. Each is described in this section.

1. INIT
2. MEDIA CHECK
3. BUILD BPB
4. READ or WRITE or WRITE TIL BUSY or WRITE WITH VERIFY or IOCTL Read or IOCTL Write
5. NON DESTRUCTIVE READ NO WAIT
6. OPEN or CLOSE (3.x)
7. REMOVABLE MEDIA (3.x)
8. STATUS
9. FLUSH

All strategy routines are called with ES:BX pointing to the Request Header. The interrupt routines get the pointers to the Request Header from the queue that the strategy routines store them in. The command code in the request header tells the driver which function to perform and what data follows the request header.

> **Note**
>
> All DWORD pointers are stored offset first, then segment.

2.7.1 INIT

Command code = 0

INIT - ES:BX ->

13-BYTE Request header
BYTE Number of units
DWORD End Address
DWORD Pointer to BPB array (Not set by character devices)
BYTE Block device number

One of the functions defined for each device driver is INIT. This routine is called only once when the device is installed. The INIT routine must return the END ADDRESS, which is a DWORD pointer to the end of the portion of the device driver to remain resident. This pointer method can be used to delete initialization code that is only needed once, saving space.

The number of units, end address, and BPB pointer are to be set by the driver. However, on entry for installable device drivers, the DWORD that is to be set by the driver to the BPB array (on block devices) points to the character after the "=" on the line in CONFIG.SYS that caused this device driver to be loaded. This allows drivers to scan the CONFIG.SYS invocation line for parameters which might be passed to the driver. This line is terminated by a Return or a Line Feed. This data is read-only and allows the device to scan the CONFIG.SYS line for arguments.

```
device=\dev\vt52.sys /1
       |
       |_____BPB address points here
```

Also, for block devices only, the drive number assigned to the first unit defined by this driver (A=0) as contained in the block device number field. This is also read-only.

For installable character devices, the end address parameter must be returned. This is a pointer to the first available byte of memory above the driver and may be used to throw away initialization code.

Block devices must return the following information:

1. The number of units must be returned. MS-DOS uses this to determine logical device names. If the current maximum logical device letter is F at the time of the install call, and the INIT routine returns 4 as the number of units, then they will have logical names G, H, I and J. This mapping is determined by the position of the driver in the device list, and by the number of units on the device (stored in the first byte of the device name field).

2. A DWORD pointer to an array of word offsets (pointers) to BPBs (BIOS Parameter Blocks) must be returned. The BPBs passed by the device driver are used by MS-DOS to create an internal structure. There must be one entry in this array for each unit defined by the device driver. In this way, if all units are the same, all of the pointers can point to the same BPB, saving space. If the device driver defines two units, then the DWORD pointer points to the first of two one-word offsets which in turn point to BPBs. The format of the BPB is described later in this chapter in Section 2.7.3, "BUILD BPB."

   Note that this array of word offsets must be protected (below the free pointer set by the return) since an internal DOS structure will be built starting at the byte pointed to by the free pointer. The defined sector size must be less than or equal to the maximum sector size defined by the resident device drivers (BIOS) during initialization. If it isn't, the installation will fail.

3. The last thing that INIT of a block device must pass back is the media descriptor byte. This byte means nothing to MS-DOS, but is passed to devices

so that they know what parameters MS-DOS is currently using for a particular drive unit.

Block devices may be either dumb or smart. A dumb device defines a unit (and therefore an internal DOS structure) for each possible media-drive combination. For example, unit 0 = drive 0 single sided, unit 1 = drive 0 double sided. For this approach, media descriptor bytes do not mean anything. A smart device allows multiple media per unit. In this case, the BPB table returned upon INIT must define sufficient space to accommodate the largest possible media supported. Smart drivers will use the media descriptor byte to pass information about what media is currently in a unit.

For more information on the media descriptor byte, see Section 2.8, "Media Descriptor Byte."

Note

If there are multiple device drivers in a single file, the ending address returned by the last INIT called will be the one MS-DOS uses. It is recommended that all of the device drivers in a single file return the same ending address. The code to remain resident for all the devices in a single file should be grouped together low in memory with the initialization code for all devices following it in memory.

2.7.2 MEDIA CHECK

Command Code = 1

MEDIA CHECK - ES:BX ->

13-BYTE Request header
BYTE Media descriptor from BPB
BYTE Returned
Returned DWORD pointer to previous Volume ID if bit 11 set and Disk Changed is returned

The MEDIA CHECK function is used with block devices only. It is called when there is a pending drive access call other than a file read or write, such as open, close, delete and rename. Its purpose is to determine whether the media in the drive has been changed. If the driver can assure that the media has not been changed (through a door-lock or other interlock mechanism), MS-DOS performance is enhanced because MS-DOS does not need to reread the FAT and invalidate in-memory buffers for each directory access.

When such a disk access call to the DOS occurs (other than a file read or write), the following sequence of events takes place:

1. The DOS converts the drive letter into a unit number of a particular block device.

2. The device driver is then called to request a media check on that subunit to see if the disk might have been changed. MS-DOS passes the old media descriptor byte. The driver returns:

   Media not changed...... (1)
   Don't know if changed...(0)
   Media changed...........(-1)
   Error

   If the media has not been changed, MS-DOS proceeds with the disk access.

   If the value returned is "Don't know," then if there are any disk sectors that have been modified and not written back out to the disk yet for this unit, MS-DOS assumes that the disk has not been changed and proceeds. MS-DOS invalidates any other buffers for the unit and does a BUILD BPB device call (see step 3, below).

   If the media has been changed, MS-DOS invalidates all buffers associated with this unit including buffers with modified data that are waiting to be written, and requests a new BIOS Parameter Block via the BUILD BPB call (see step 3, below).

3. Once the BPB has been returned, MS-DOS corrects its internal structure for the drive from the new BPB and proceeds with the access after reading the directory and the FAT.

Note that the previous media ID byte is passed to the device driver. If the old media ID byte is the same as the new one, the disk might have been changed and a new disk may be in the drive; therefore, all FAT, directory, and data sectors that are buffered in memory for the unit are considered to be invalid.

If the driver has bit 11 of the device attribute word set to 1, and the driver returns -1, Media Changed, the driver must set the DWORD pointer to the previous Volume ID field. If the DOS determines that Media Changed is an error based on the state of the DOS buffer cache, the DOS will generate a 0FH error on behalf of the device. If the driver does not implement Volume ID support, but has bit 11 set, (it should set a static pointer to the string "NO NAME",0.)

A creative solution to the problem of no door-locks follows:

It has been determined that it is impossible for a user to change a disk in less than 2 seconds; therefore, when MEDIA CHECK occurs within 2 seconds of a disk access, the driver reports "1," "Media not changed." This makes a tremendous improvement in performance.

Note

If the media ID byte in the returned BPB is the same as the previous media ID byte, MS-DOS will assume that the format of the disk is the same (even though the disk may have been changed) and will skip the step of updating its internal structure. Therefore, all BPBs must have unique media bytes regardless of FAT ID bytes.

2.7.3 BUILD BPB (BIOS Parameter Block)

Command code = 2

BUILD BPB - ES:BX ->

13-BYTE Request header
BYTE Media descriptor from BPB
DWORD Transfer address (Points to one sector worth of scratch space or first sector of FAT depending on the value of Bit 13 in the device attribute word.)
DWORD Pointer to BPB

The Build BPB function is used with block devices only. As described in the MEDIA CHECK function, the BUILD BPB function will be called any time that a preceding MEDIA CHECK call indicates that the disk has been or might have been changed. The device driver must return a pointer to a BPB. This is different from the INIT call where a pointer to an array of word offsets to BPBs is returned.

The BUILD BPB call gets a DWORD pointer to a one-sector buffer. The contents of this buffer are determined by the NON FAT ID bit (bit 13) in the attribute field. If the bit is zero, then the buffer contains the first sector of the first FAT. The FAT ID byte is the first byte of this buffer. In this case, the driver must not alter this buffer. Note that the location of the FAT must be the same for all possible media because this first FAT sector must be read <u>before</u> the actual BPB is returned. If the NON FAT ID bit is set, then the pointer points to one sector of scratch space (which may be used for anything). Refer to Section 2.8, "Media Descriptor Byte,"" and Section 2.9, "Format of a Media Descriptor Table," for information on how to construct the BPB.

MS-DOS 3.x includes additional support for devices that have door-locks or some other means of telling when a disk has been changed. There is a new error that can be returned from the device driver (error 15). The error means "the disk has been changed when it shouldn't have been," and the user is prompted for the correct disk using a Volume ID. The driver may generate this error on read or write. The DOS may generate the error on MEDIA CHECK if the driver reports media changed, and there are buffers in the DOS buffer cache that need to be flushed to the previous disk.

For drivers that support this error, the BUILD BPB function is a trigger that causes a new Volume ID to be read off the disk. This action indicates that the disk has been legally changed. A Volume ID is placed on a disk by the FORMAT utility, and is simply an entry in the root directory of the disk that has the Volume ID attribute. It is stored by the driver as an ASCIZ string.

The requirement that the driver return a Volume ID does not exclude some other Volume identifier scheme as long as the scheme uses ASCIZ strings. A NUL (nonexistent or unsupported) Volume ID is by convention the string:

```
     DB      "NO NAME     ",0
```

2.7.4 READ or WRITE

Command codes = 3,4,8,9, 12, and 16

READ OR WRITE (Including IOCTL) or
   OUTPUT UNTIL BUSY - ES:BX ->

13-BYTE Request header
BYTE Media descriptor from BPB
DWORD Transfer address
WORD Byte/sector count
WORD Starting sector number (Ignored on character devices)
Returned DWORD pointer to requested Volume ID if error 0FH

COMMAND CODE	REQUEST
3	IOCTL READ
4	READ (block or character)
8	WRITE (block or character)
9	WRITE WITH VERIFY
12	IOCTL WRITE
16	OUTPUT TIL BUSY (char devs only)

The driver must perform the READ or WRITE call depending on which command code is set. Block devices read or write sectors; character devices read or write bytes.

When I/O completes, the device driver must set the status word and report the number of sectors or bytes successfully transferred. This should be done even if an error prevented the transfer from being completed. Setting the error bit and error code alone is not sufficient.

In addition to setting the status word, the driver must set the sector count to the actual number of sectors (or bytes) transferred. No error check is performed on an IOCTL I/O call. The device driver must always set the return byte/sector count to the actual number of bytes/sectors successfully transferred.

If the verify switch is on, the device driver will be called with command code 9 (WRITE WITH VERIFY). Your device driver will be responsible for verifying the write.

If the driver returns error code 0FH (Invalid disk change), it must return a DWORD pointer to an ASCIZ string (which is the correct Volume ID). Returning this error code triggers the DOS to prompt the user to re-insert the disk. The device driver should have read the Volume ID as a result of the BUILD BPB function.

Drivers may maintain a reference count of open files on the disk by monitoring the OPEN and CLOSE functions. This allows the driver to determine when to return error 0FH. If there are no open files (reference count = 0), and the disk has been changed, the I/O is okay. If there are open files, however, an 0FH error may exist.

The OUTPUT UNTIL BUSY call is a speed optimization on character devices only for print spoolers. The device driver is expected to output all the characters possible until the device returns busy. Under no circumstances should the device driver block during this function. Note that it is not an error for the device driver to return the number of bytes output being less than the number of bytes requested (or = 0).

The OUTPUT UNTIL BUSY call allows spooler programs to take advantage of the burst behavior of most printers. Many printers have on-board RAM buffers which typically hold a line or a fixed amount of characters. These buffers fill up without the printer going busy, or going busy for a very short period (less than 10 instructions) between characters. A line of characters can be very quickly output to the printer, then the printer is busy for a long time while the characters are being printed. This new device call allows background spooling programs to use this burst behavior efficiently. Rather than take the overhead of a device driver call for each character, or risk getting stuck in the device driver outputting a block of characters, this call allows a burst of characters to be output without the device driver having to wait for the device to be ready.

THE FOLLOWING APPLIES TO BLOCK DEVICE DRIVERS:

Under certain circumstances, the BIOS may be asked to perform a write operation of 64K bytes, which seems to be a "wrap around" of the transfer address in the BIOS I/O packet. This request arises due to an optimization added to the write code in MS-DOS. It will only manifest on user writes that are within a sector size of 64K bytes on files "growing" past the current EOF. It is allowable for the BIOS to ignore the balance of the write that "wraps around" if it so chooses. For example, a write of 10000H bytes worth of sectors with a transfer address of XXX:1 could ignore the last two bytes. A user program can never request an I/O of more than FFFFH bytes and cannot wrap around (even to 0) in the transfer segment. Therefore, in this case, the last two bytes can be ignored.

MS-DOS maintains two FATs. If the DOS has problems reading the first, it automatically tries the second before reporting the error. The BIOS is responsible for all retries.

Although the COMMAND.COM handler does no automatic retries, there are applications that have their own Interrupt 24H handlers that do automatic retries on certain types of Interrupt 24H errors before reporting them.

2.7.5 NON DESTRUCTIVE READ NO WAIT

Command code = 5

NON DESTRUCTIVE READ NO WAIT - ES:BX ->

13-BYTE Request header
BYTE read from device

This call allows MS-DOS to look ahead one input character. The device sets the done bit in the status word.

If the character device returns busy bit = 0 (there are characters in the buffer), then the next character that would be read is returned. This character is not removed from the input buffer (hence the term "Non Destructive Read"). If the character device returns busy bit = 1, there are no characters in the buffer.

2.7.6 OPEN or CLOSE

Command codes = 13 and 14

OPEN or CLOSE - ES:BX ->

13-BYTE Static request header

These functions are only called by MS-DOS 3.x if the device driver sets the OPEN/CLOSE/RM attribute bit in the device header. They are designed to inform the device about current file activity on the device. On block devices, they can be used to manage local buffering. The device can keep a reference count. Every OPEN causes the device to increment the count, every CLOSE to decrement. When the count goes to zero; it means there are no open files on the device, and the device should flush any buffers that have been written to that may have been used inside the device because it is now "legal" for the user to change the media on a removable media drive.

There are problems with this mechanism on block devices because programs that use FCB calls can open files without closing them. It is therefore advisable to reset the count to zero without flushing the buffers when the answer to "has the media been changed?" is yes and the BUILD BPB call is made to the device.

These calls are of more use on character devices. The OPEN call can be used to send a device initialization string. On a printer, this could cause a string for setting font and page size characteristics to be sent to the printer so that it would always be in a known state at the start of an I/O stream. Using IOCTL to set these pre- and post-strings provides a flexible mechanism of serial I/O device stream control. The reference count mechanism can also be used to detect a simultaneous access error. It may be desirable to disallow more than one OPEN on a device at any given time. In this case, a second OPEN would result in an error.

Note that since all processes have access to stdin, stdout, stderr, stdaux, and stdprn (handles 0,1,2,3,4), the CON, AUX, and PRN devices are always open.

2.7.7 REMOVABLE MEDIA

Command code = 15

REMOVABLE MEDIA - ES:BX ->

13-BYTE Static request header

This function is only called by MS-DOS 3.x if the device driver sets the OPEN/CLOSE/RM attribute bit in the device header. This call is given only to block devices by a subfunction of the IOCTL system call. It is sometimes desirable for a utility to know whether it is dealing with a non-removable media drive (such as a hard disk), or a removable media drive (like a floppy). An example is the FORMAT utility which prints different versions of some of the prompts.

The information is returned in the busy bit of the status word. If the busy bit is 1, then the media is non-removable. If the busy bit is 0, then the media is removable. Note that no checking of the error bit is performed. It is assumed that this call always succeeds.

2.7.8 STATUS

Command codes = 6 and 10

STATUS Calls ES:BX ->

13-BYTE request header

This call returns information to the DOS as to whether data is waiting for input or output. All the driver must do is set the status word and the busy bit as follows:

> <u>For output on character devices</u>: If the driver sets bit 9 to 1 on return, it informs the DOS that a write request (if made) would wait for completion of a current request. If it is 0, there is no current request and a write request (if made) would start immediately.
>
> <u>For input on character devices with a buffer</u>: A return of 1 implies that no characters are buffered and that a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the device buffer and a read would not be blocked. A return of 0 implies that

the user has typed something. MS-DOS assumes that all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy = 0 so that the DOS will not hang waiting for something to get into a non-existent buffer.

2.7.9 FLUSH

Command codes = 7 and 11

FLUSH Calls - ES:BX ->

13-BYTE request header

The FLUSH call tells the driver to flush (terminate) all pending requests. This call is used to flush the input queue on character devices.

The device driver performs the flush function, sets the status word, and returns.

2.8 MEDIA DESCRIPTOR BYTE

In MS-DOS, the media descriptor byte is used to inform the DOS that a different type of media is present. The media descriptor byte can be any value between 0 and FFH. It does not have to be the same as the FAT ID byte. The FAT ID byte, which is the first byte of the FAT, was used in MS-DOS 1.00 to distinguish between different types of disk media and may be used as well under 2.x and 3.x disk device drivers. However, FAT ID bytes only have significance for block device drivers where the NON FAT ID bit is not set (0).

Values of the media descriptor byte or the FAT ID byte have no significance to MS-DOS. They are passed to the device driver to facilitate media determination in any way the OEM chooses to implement.

Important

When the BPB call is made, if the media byte returned in the new BPB is the same as the old media byte, the DOS does not rebuild its internal structure for the device. MS-DOS will treat the disk as though the format has not changed, even though the physical disk might have changed. Therefore, each BPB must have a unique media descriptor byte.

2.9 FORMAT OF A MEDIA DESCRIPTOR TABLE

The MS-DOS file system uses a linked list of pointers (one for each cluster or allocation unit) called the File Allocation Table (FAT). Unused clusters are represented by zero and end of file by FFF (or FFFF on units with 16-bit FAT entries). No valid entry should ever point to a zero entry, but if it does, the first FAT entry (which would be pointed to by a zero entry) was reserved and set to end of chain. Eventually, several end of chain values were defined ([F]FF8-[F]FFF), and these were used to distinguish different types of media.

A preferrable technique is to write a complete media descriptor table in the boot sector and use it for media identification. To ensure backward compatibility for systems whose drivers do not set the NON FAT ID bit (including the IBM PC implementation), it is necessary also to write the FAT ID bytes during the FORMAT process.

To allow more flexibility for supporting many different disk formats in the future, it is recommended that the information relating to the BPB for a particular piece of media be kept in the boot sector. Figure 2.3 shows the format of such a boot sector.

	3 BYTE Near JUMP to boot code
	8 BYTES OEM name and version
B P	WORD Bytes per sector
B	BYTE Sectors per allocation unit
↓	WORD Reserved sectors
	BYTE Number of FATs
	WORD Number of root dir entries
↑	WORD Number of sectors in logical image
B P	BYTE Media descriptor
B	WORD Number of FAT sectors
	WORD Sectors per track
	WORD Number of heads
	WORD Number of hidden sectors

Figure 2.3. Format of Boot Sector

The three words at the end ("Sectors per track," "Number of heads," and "Number of hidden sectors") are not used by the DOS but may be used by device drivers. They are intended to help the device driver understand the media. "Sectors per track" and "Number of heads" are useful for supporting different media which may have the same logical layout but a different physical layout (e.g., 40 track, double-sided versus 80 track, single-sided). "Sectors per track" tells the device driver how the logical disk format is laid out on the physical disk. "Number of hidden sectors" may be used to support drive-partitioning schemes.

The following procedure is recommended for media determination by NON FAT ID format drivers:

1. Read the boot sector of the drive into the 1-sector scratch space pointed to by the DWORD Transfer address.

2. Determine if the first byte of the boot sector is an E9H or EBIT (the first byte of a 3-byte NEAR or 2-byte short jump) or an EBH (the first byte of a

2-byte jump followed by a NOP). If so, a BPB is located beginning at offset 3. Return a pointer to it.

3. If the boot sector does not have a BPB table, it probably is a disk formatted under a version 1.x implementation of MS-DOS and probably uses a FAT ID byte for media determination.

   The driver may optionally attempt to read the first sector of the FAT into the 1-sector scratch area and read the first byte to determine media type based upon whatever FAT ID bytes may have been used on disks that are expected to be read by this system. Return a pointer to a hard-coded BPB.

2.10 THE CLOCK DEVICE

MS-DOS assumes that some sort of clock is available in the system. This may either be a CMOS real-time clock or an interval timer which is initialized at boot time by the user. The CLOCK device defines and performs functions like any other character device except that it is identified by a bit in the attribute word. The DOS uses this bit to identify it and consequently this device may take any name. The IBM implementation uses "$CLOCK" so as not to conflict with existing files named "CLOCK."

The CLOCK device is unique in that MS-DOS will read or write a 6-byte sequence which encodes the date and time. A write to this device will set the date and time, and a read will get the date and time.

Figure 2.4 illustrates the binary time format used by the CLOCK device:

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5
days since 1-1-80 low byte	hi byte	minutes	hours	sec/100	seconds

Figure 2.4 CLOCK Device Format

2.11 ANATOMY OF A DEVICE CALL

The following steps illustrate what happens when MS-DOS calls on a block device driver to perform a WRITE request:

1. MS-DOS writes a request packet in a reserved area of memory.

2. MS-DOS calls the block device driver strategy entry point.

3. The device driver saves the ES and BX registers (ES:BX points to the request packet) and does a FAR return.

4. MS-DOS calls the interrupt entry point.

5. The device driver retrieves the pointer to the request packet and reads the command code (offset 2) to determine that this is a write request. The device driver converts the command code to an index into a dispatch table and control passes to the disk write routine.

6. The device driver reads the unit code (offset 1) to determine to which disk drive it is supposed to write.

7. Since the command is a disk write, the device driver must get the transfer address (offset 14), the sector count (offset 18), and the start sector (offset 20) in the request packet.

8. The device driver translates the first logical sector number into a track, head, and sector number.

9. The device driver writes the specified number of sectors, starting at the beginning sector on the drive defined by the unit code (the subunit defined by this device driver), and transfers data from the transfer address indicated in the request packet. Note that this may involve multiple write commands to the disk controller.

10. After the transfer is complete, the device driver must report the status of the request to MS-DOS by setting the done bit in the status word (offset 3 in the request packet). It reports the number of sectors actually transferred in the sector count area of the request packet.

11. If an error occurs, the driver sets the done bit and the error bit in the status word and fills in the error code in the lower half of the status word. The number of sectors actually transferred must be written in the request header. It is not sufficient just to set the error bit of the status word.

12. The device driver does a FAR return to MS-DOS.

The device drivers should preserve the state of MS-DOS. This means that all registers (including flags) should be preserved. The direction flag and interrupt enable bits are critical. When the interrupt entry point in the device driver is called, MS-DOS has room for about 40 to 50 bytes on its internal stack. Your device driver should switch to a local stack if it uses extensive stack operations.

2.12 EXAMPLE OF DEVICE DRIVERS

The following examples illustrate a block device driver and a character device driver program.

2.12.1 Block Device Driver

```
;********************* A BLOCK DEVICE ******************

        TITLE   5 1/4" DISK DRIVER FOR SCP DISK-MASTER

;This driver is intended to drive up to four 5 1/4" drives
;hooked to the Seattle Computer Products DISK MASTER disk
;controller. All standard IBM PC formats are supported.


FALSE   EQU     0
TRUE    EQU     NOT FALSE

;The I/O port address of the DISK MASTER
DISK    EQU     0E0H
;DISK+0
;       1793    Command/Status
;DISK+1
;       1793    Track
;DISK+2
;       1793    Sector
;DISK+3
;       1793    Data
;DISK+4
;       Aux Command/Status
;DISK+5
;       Wait Sync

;Back side select bit
BACKBIT EQU     04H
;5 1/4" select bit
SMALBIT EQU     10H
;Double Density bit
DDBIT   EQU     08H

;Done bit in status register
DONEBIT EQU     01H

;Use table below to select head step speed.
;Step times for 5" drives
;are double that shown in the table.
;
;Step value    1771    1793
;
;    0          6ms     3ms
```

```
;       1           6ms      6ms
;       2          10ms     10ms
;       3          20ms     15ms
;
STPSPD  EQU         1

NUMERR  EQU         ERROUT-ERRIN

CR      EQU         0DH
LF      EQU         0AH

CODE    SEGMENT
ASSUME  CS:CODE,DS:NOTHING,ES:NOTHING,SS:NOTHING
;-------------------------------------------------------
;
;       DEVICE HEADER
;
DRVDEV  LABEL       WORD
        DW          -1,-1
        DW          0000     ;IBM format-compatible, Block
        DW          STRATEGY
        DW          DRV$IN
DRVMAX  DB          4

DRVTBL  LABEL       WORD
        DW          DRV$INIT
        DW          MEDIA$CHK
        DW          GET$BPB
        DW          CMDERR
        DW          DRV$READ
        DW          EXIT
        DW          EXIT
        DW          EXIT
        DW          DRV$WRIT
        DW          DRV$WRIT
        DW          EXIT
        DW          EXIT
        DW          EXIT

;-----------------------------------
;
;       STRATEGY

PTRSAV  DD          0

STRATP  PROC        FAR
STRATEGY:
        MOV         WORD PTR [PTRSAV],BX
        MOV         WORD PTR [PTRSAV+2],ES
        RET
STRATP  ENDP

;-------------------------------------
;
;       MAIN ENTRY
```

```
CMDLEN  =       0           ;LENGTH OF THIS COMMAND
UNIT    =       1           ;SUB UNIT SPECIFIER
CMDC    =       2           ;COMMAND CODE
STATUS  =       3           ;STATUS
MEDIA   =       13          ;MEDIA DESCRIPTOR
TRANS   =       14          ;TRANSFER ADDRESS
COUNT   =       18          ;COUNT OF BLOCKS OR CHARACTERS
START   =       20          ;FIRST BLOCK TO TRANSFER

DRV$IN:
        PUSH    SI
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        PUSH    BX

        LDS     BX,[PTRSAV]     ;GET POINTER TO I/O PACKET

        MOV     AL,BYTE PTR [BX].UNIT   ;AL = UNIT CODE
        MOV     AH,BYTE PTR [BX].MEDIA  ;AH = MEDIA DESCRIP
        MOV     CX,WORD PTR [BX].COUNT  ;CX = COUNT
        MOV     DX,WORD PTR [BX].START  ;DX = START SECTOR
        PUSH    AX
        MOV     AL,BYTE PTR [BX].CMDC   ;Command code
        CMP     AL,15
        JA      CMDERRP                 ;Bad command
        CBW
        SHL     AX,1                    ;2 times command =
                                        ;word table index
        MOV     SI,OFFSET DRVTBL
        ADD     SI,AX                   ;Index into table
        POP     AX                      ;Get back media
                                        ;and unit

        LES     DI,DWORD PTR [BX].TRANS ;ES:DI = TRANSFER
                                        ;ADDRESS

        PUSH    CS
        POP     DS

ASSUME  DS:CODE

        JMP     WORD PTR [SI]             ;GO DO COMMAND

;-----------------------------------------------------------
;
;       EXIT - ALL ROUTINES RETURN THROUGH THIS PATH
;
ASSUME  DS:NOTHING
```

```
CMDERRP:
        POP     AX                      ;Clean stack
CMDERR:
        MOV     AL,3                    ;UNKNOWN COMMAND ERROR
        JMP     SHORT ERR$EXIT

ERR$CNT:LDS     BX,[PTRSAV]
        SUB     WORD PTR [BX].COUNT,CX ;# OF SUCCESS. I/Os

ERR$EXIT:
;AL has error code
        MOV     AH,10000001B             ;MARK ERROR RETURN
        JMP     SHORT ERR1

EXITP   PROC    FAR

EXIT:   MOV     AH,00000001B
ERR1:   LDS     BX,[PTRSAV]
        MOV     WORD PTR [BX].STATUS,AX
                                      ;MARK OPERATION COMPLETE

        POP     BX
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     DX
        POP     CX
        POP     AX
        POP     SI
        RET                           ;RESTORE REGS AND RETURN
EXITP   ENDP

CURDRV  DB      -1

TRKTAB  DB      -1,-1,-1,-1

SECCNT  DW      0

DRVLIM  =       8       ;Number of sectors on device
SECLIM  =       13      ;MAXIMUM SECTOR
HDLIM   =       15      ;MAXIMUM HEAD

;WARNING - preserve order of drive and curhd!

DRIVE   DB      0       ;PHYSICAL DRIVE CODE
CURHD   DB      0       ;CURRENT HEAD
CURSEC  DB      0       ;CURRENT SECTOR
CURTRK  DW      0       ;CURRENT TRACK


;
MEDIA$CHK:              ;Always indicates Don't know
ASSUME  DS:CODE
        TEST    AH,00000100B        ;TEST IF MEDIA REMOVABLE
```

```
        JZ      MEDIA$EXT
        XOR     DI,DI                   ;SAY I DON'T KNOW
MEDIA$EXT:
        LDS     BX,[PTRSAV]
        MOV     WORD PTR [BX].TRANS,DI
        JMP     EXIT

BUILD$BPB:
ASSUME  DS:CODE
        MOV     AH,BYTE PTR ES:[DI]         ;GET FAT ID BYTE
        CALL    BUILDBP                     ;TRANSLATE
SETBPB: LDS     BX,[PTRSAV]
        MOV     [BX].MEDIA,AH
        MOV     [BX].COUNT,DI
        MOV     [BX].COUNT+2,CS
        JMP     EXIT

BUILDBP:
ASSUME  DS:NOTHING
;AH is media byte on entry
;DI points to correct BPB on return
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    BX
        MOV     CL,AH         ;SAVE MEDIA
        AND     CL,0F8H       ;NORMALIZE
        CMP     CL,0F8H       ;COMPARE WITH GOOD MEDIA BYTE
        JZ      GOODID
        MOV     AH,0FEH       ;DEFAULT TO 8-SECTOR,
                              ;SINGLE-SIDED
GOODID:
        MOV     AL,1          ;SET NUMBER OF FAT SECTORS
        MOV     BX,64*256+8 ;SET DIR ENTRIES AND SECTOR MAX
        MOV     CX,40*8       ;SET SIZE OF DRIVE
        MOV     DX,01*256+1 ;SET HEAD LIMIT & SEC/ALL UNIT
        MOV     DI,OFFSET DRVBPB
        TEST    AH,00000010B ;TEST FOR 8 OR 9 SECTOR
        JNZ     HAS8          ;NZ = HAS 8 SECTORS
        INC     AL            ;INC NUMBER OF FAT SECTORS
        INC     BL            ;INC SECTOR MAX
        ADD     CX,40         ;INCREASE SIZE
HAS8:   TEST    AH,00000001B    ;TEST FOR 1 OR 2 HEADS
        JZ      HAS1          ;Z = 1 HEAD
        ADD     CX,CX         ;DOUBLE SIZE OF DISK
        MOV     BH,112        ;INCREASE # OF DIREC. ENTRIES
        INC     DH            ;INC SEC/ALL UNIT
        INC     DL            ;INC HEAD LIMIT
HAS1:   MOV     BYTE PTR [DI].2,DH
        MOV     BYTE PTR [DI].6,BH
        MOV     WORD PTR [DI].8,CX
        MOV     BYTE PTR [DI].10,AH
        MOV     BYTE PTR [DI].11,AL
        MOV     BYTE PTR [DI].13,BL
        MOV     BYTE PTR [DI].15,DL
```

```
        POP     BX
        POP     DX
        POP     CX
        POP     AX
        RET


;------------------------------------------------------------
;
;       DISK I/O HANDLERS
;
;ENTRY:
;       AL = DRIVE NUMBER (0-3)
;       AH = MEDIA DESCRIPTOR
;       CX = SECTOR COUNT
;       DX = FIRST SECTOR
;       DS = CS
;       ES:DI = TRANSFER ADDRESS
;EXIT:
;       IF SUCCESSFUL CARRY FLAG = 0
;         ELSE CF=1 AND AL CONTAINS (MS-DOS) ERROR CODE,
          CX # sectors NOT transferred

DRV$READ:
ASSUME  DS:CODE
        JCXZ    DSKOK
        CALL    SETUP
        JC      DSK$IO
        CALL    DISKRD
        JMP     SHORT DSK$IO

DRV$WRIT:
ASSUME  DS:CODE
        JCXZ    DSKOK
        CALL    SETUP
        JC      DSK$IO
        CALL    DISKWRT
ASSUME  DS:NOTHING
DSK$IO: JNC     DSKOK
        JMP     ERR$CNT
DSKOK:  JMP     EXIT


SETUP:
ASSUME  DS:CODE
;Input same as above
;On output
; ES:DI = Trans addr
; DS:BX Points to BPB
; Carry set if error (AL is error code (MS-DOS))
; else
;        [DRIVE]  = Drive number (0-3)
;        [SECCNT]  = Sectors to transfer
;        [CURSEC]  = Sector number of start of I/O
;        [CURHD]   = Head number of start of I/O    ;Set
```

```
;       [CURTRK] = Track # of start of I/O ;Seek performed
; All other registers destroyed

        XCHG    BX,DI                   ;ES:BX = TRANSFER ADDRESS
        CALL    BUILDBP                 ;DS:DI = PTR TO B.P.B
        MOV     SI,CX
        ADD     SI,DX
        CMP     SI,WORD PTR [DI].DRVLIM
                                       ;COMPARE AGAINST DRIVE MAX
        JBE     INRANGE
        MOV     AL,8
        STC
        RET

INRANGE:
        MOV     [DRIVE],AL
        MOV     [SECCNT],CX        ;SAVE SECTOR COUNT
        XCHG    AX,DX              ;SET UP LOGICAL SECTOR
                                   ;FOR DIVIDE
        XOR     DX,DX
        DIV     WORD PTR [DI].SECLIM ;DIVIDE BY SEC PER TRACK
        INC     DL
        MOV     [CURSEC],DL            ;SAVE CURRENT SECTOR
        MOV     CX,WORD PTR [DI].HDLIM ;GET NUMBER OF HEADS
        XOR     DX,DX    ;DIVIDE TRACKS BY HEADS PER CYLINDER
        DIV     CX
        MOV     [CURHD],DL         ;SAVE CURRENT HEAD
        MOV     [CURTRK],AX        ;SAVE CURRENT TRACK
SEEK:
        PUSH    BX                 ;Xaddr
        PUSH    DI                 ;BPB pointer
        CALL    CHKNEW             ;Unload head if change drives
        CALL    DRIVESEL
        MOV     BL,[DRIVE]
        XOR     BH,BH              ;BX drive index
        ADD     BX,OFFSET TRKTAB          ;Get current track
        MOV     AX,[CURTRK]
        MOV     DL,AL            ;Save desired track
        XCHG    AL,DS:[BX]       ;Make desired track current
        OUT     DISK+1,AL        ;Tell Controller current track
        CMP     AL,DL            ;At correct track?
        JZ      SEEKRET          ;Done if yes
        MOV     BH,2             ;Seek retry count
        CMP     AL,-1            ;Position Known?
        JNZ     NOHOME           ;If not home head
TRYSK:
        CALL    HOME
        JC      SEEKERR
NOHOME:
        MOV     AL,DL
        OUT     DISK+3,AL          ;Desired track
        MOV     AL,1CH+STPSPD      ;Seek
        CALL    DCOM
        AND     AL,98H     ;Accept not rdy, seek, & CRC errors
        JZ      SEEKRET
```

```
        JS      SEEKERR         ;No retries if not ready
        DEC     BH
        JNZ     TRYSK
SEEKERR:
        MOV     BL,[DRIVE]
        XOR     BH,BH           ;BX drive index
        ADD     BX,OFFSET TRKTAB        ;Get current track
        MOV     BYTE PTR DS:[BX],-1     ;Make current track
                                        ;unknown
        CALL    GETERRCD
        MOV     CX,[SECCNT]     ;Nothing transferred
        POP     BX              ;BPB pointer
        POP     DI              ;Xaddr
        RET

SEEKRET:
        POP     BX              ;BPB pointer
        POP     DI              ;Xaddr
        CLC
        RET

;----------------------------------------------
;
;       READ
;

DISKRD:
ASSUME  DS:CODE
        MOV     CX,[SECCNT]
RDLP:
        CALL    PRESET
        PUSH    BX
        MOV     BL,10              ;Retry count
        MOV     DX,DISK+3          ;Data port
RDAGN:
        MOV     AL,80H             ;Read command
        CLI                        ;Disable for 1793
        OUT     DISK,AL            ;Output read command
        MOV     BP,DI              ;Save address for retry
        JMP     SHORT RLOOPENTRY
RLOOP:
        STOSB
RLOOPENTRY:
        IN      AL,DISK+5          ;Wait for DRQ or INTRQ
        SHR     AL,1
        IN      AL,DX              ;Read data
        JNC     RLOOP
        STI                        ;Ints OK now
        CALL    GETSTAT
        AND     AL,9CH
        JZ      RDPOP              ;Ok
        MOV     DI,BP              ;Get back transfer
        DEC     BL
        JNZ     RDAGN
        CMP     AL,10H             ;Record not found?
```

```
        JNZ     GOT_CODE                ;No
        MOV     AL,1                    ;Map it
GOT_CODE:
        CALL    GETERRCD
        POP     BX
        RET

RDPOP:
        POP     BX
        LOOP    RDLP
        CLC
        RET


;-----------------------------------------------
;
;       WRITE
;

DISKWRT:
ASSUME  DS:CODE
        MOV     CX,[SECCNT]
        MOV     SI,DI
        PUSH    ES
        POP     DS
ASSUME  DS:NOTHING
WRLP:
        CALL    PRESET
        PUSH    BX
        MOV     BL,10                   ;Retry count
        MOV     DX,DISK+3               ;Data port
WRAGN:
        MOV     AL,0A0H                 ;Write command
        CLI                             ;Disable for 1793
        OUT     DISK,AL                 ;Output write command
        MOV     BP,SI                   ;Save address for retry
WRLOOP:
        IN      AL,DISK+5
        SHR     AL,1
        LODSB                           ;Get data
        OUT     DX,AL                   ;Write data
        JNC     WRLOOP
        STI                             ;Ints OK now
        DEC     SI
        CALL    GETSTAT
        AND     AL,0FCH
        JZ      WRPOP                   ;Ok
        MOV     SI,BP                   ;Get back transfer
        DEC     BL
        JNZ     WRAGN
        CALL    GETERRCD
        POP     BX
        RET

WRPOP:
```

```
        POP     BX
        LOOP    WRLP
        CLC
        RET


PRESET:
ASSUME  DS:NOTHING
        MOV     AL,[CURSEC]
        CMP     AL,CS:[BX].SECLIM
        JBE     GOTSEC
        MOV     DH,[CURHD]
        INC     DH
        CMP     DH,CS:[BX].HDLIM
        JB      SETHEAD                 ;Select new head
        CALL    STEP                    ;Go on to next track
        XOR     DH,DH                   ;Select head zero
SETHEAD:
        MOV     [CURHD],DH
        CALL    DRIVESEL
        MOV     AL,1                    ;First sector
        MOV     [CURSEC],AL             ;Reset CURSEC
GOTSEC:
        OUT     DISK+2,AL          ;Tell controller which sector
        INC     [CURSEC]           ;We go on to next sector
        RET

STEP:
ASSUME  DS:NOTHING
        MOV     AL,58H+STPSPD   ;Step in w/ update, no verify
        CALL    DCOM
        PUSH    BX
        MOV     BL,[DRIVE]
        XOR     BH,BH             ;BX drive index
        ADD     BX,OFFSET TRKTAB          ;Get current track
        INC     BYTE PTR CS:[BX]          ;Next track
        POP     BX
        RET

HOME:
ASSUME  DS:NOTHING
        MOV     BL,3
TRYHOM:
        MOV     AL,0CH+STPSPD     ;Restore with verify
        CALL    DCOM
        AND     AL,98H
        JZ      RET3
        JS      HOMERR            ;No retries if not ready
        PUSH    AX                ;Save real error code
        MOV     AL,58H+STPSPD     ;Step in w/ update no verify
        CALL    DCOM
        DEC     BL
        POP     AX                ;Get back real error code
        JNZ     TRYHOM
HOMERR:
```

```
        STC
RET3:   RET


CHKNEW:
ASSUME  DS:NOTHING
        MOV     AL,[DRIVE]          ;Get disk drive number
        MOV     AH,AL
        XCHG    AL,[CURDRV]         ;Make new drive current.
        CMP     AL,AH               ;Changing drives?
        JZ      RET1                ;No
; If changing drives, unload head so the head load delay
;one-shot will fire again. Do it by seeking to the same
;track with the H bit reset.
;
        IN      AL,DISK+1           ;Get current track number
        OUT     DISK+3,AL           ;Make it the track to seek
        MOV     AL,10H              ;Seek and unload head

DCOM:
ASSUME  DS:NOTHING
        OUT     DISK,AL
        PUSH    AX
        AAM                         ;Delay 10 microseconds
        POP     AX
GETSTAT:
        IN      AL,DISK+4
        TEST    AL,DONEBIT
        JZ      GETSTAT
        IN      AL,DISK
RET1:   RET

DRIVESEL:
ASSUME  DS:NOTHING
;Select the drive based on current info
;Only AL altered
        MOV     AL,[DRIVE]
        OR      AL,SMALBIT + DDBIT        ;5 1/4" IBM PC disks
        CMP     [CURHD],0
        JZ      GOTHEAD
        OR      AL,BACKBIT          ;Select side 1
GOTHEAD:
        OUT     DISK+4,AL           ;Select drive and side
        RET

GETERRCD:
ASSUME  DS:NOTHING
        PUSH    CX
        PUSH    ES
        PUSH    DI
        PUSH    CS
        POP     ES                  ;Make ES the local segment
        MOV     CS:[LSTERR],AL ;Terminate list w/ error code
        MOV     CX,NUMERR           ;Number of error conditions
        MOV     DI,OFFSET ERRIN ;Point to error conditions
```

```
        REPNE   SCASB
        MOV     AL,NUMERR-1[DI] ;Get translation
        STC                     ;Flag error condition
        POP     DI
        POP     ES
        POP     CX
        RET                     ;and return

;**********************************************************
;       BPB FOR AN IBM FLOPPY DISK, VARIOUS PARAMETERS ARE
;       PATCHED BY BUILDBP TO REFLECT THE TYPE OF MEDIA
;       INSERTED
;       This is a nine sector single side BPB
DRVBPB:
        DW      512             ;Physical sector size in bytes
        DB      1               ;Sectors/allocation unit
        DW      1               ;Reserved sectors for DOS
        DB      2               ;# of allocation tables
        DW      64              ;Number directory entries
        DW      9*40            ;Number 512-byte sectors
        DB      11111100B       ;Media descriptor
        DW      2               ;Number of FAT sectors
        DW      9               ;Sector limit
        DW      1               ;Head limit


INITAB  DW      DRVBPB                  ;Up to four units
        DW      DRVBPB
        DW      DRVBPB
        DW      DRVBPB

ERRIN:  ;DISK ERRORS RETURNED FROM THE 1793 CONTROLER
        DB      80H             ;NO RESPONSE
        DB      40H             ;Write protect
        DB      20H             ;Write Fault
        DB      10H             ;SEEK error
        DB      8               ;CRC error
        DB      1               ;Mapped from 10H
                                ;(record not found) on READ
LSTERR  DB      0               ;ALL OTHER ERRORS

ERROUT: ;RETURNED ERROR CODES CORRESPONDING TO ABOVE
        DB      2               ;NO RESPONSE
        DB      0               ;WRITE ATTEMPT
                                ;ON WRITE-PROTECT DISK
        DB      0AH             ;WRITE FAULT
        DB      6               ;SEEK FAILURE
        DB      4               ;BAD CRC
        DB      8               ;SECTOR NOT FOUND
        DB      12              ;GENERAL ERROR


DRV$INIT:
;
; Determine number of physical drives by reading CONFIG.SYS
```

```
;
ASSUME   DS:CODE
         PUSH     DS
         LDS      SI,[PTRSAV]
ASSUME   DS:NOTHING
         LDS      SI,DWORD PTR [SI.COUNT] ;DS:SI points to
                                          ;CONFIG.SYS
SCAN_LOOP:
         CALL     SCAN_SWITCH
         MOV      AL,CL
         OR       AL,AL
         JZ       SCAN4
         CMP      AL,"s"
         JZ       SCAN4

WERROR:  POP      DS
ASSUME   DS:CODE
         MOV      DX,OFFSET ERRMSG2
WERROR2: MOV      AH,9
         INT      21H
         XOR      AX,AX
         PUSH     AX                      ;No units
         JMP      SHORT ABORT

BADNDRV:
         POP      DS
         MOV      DX,OFFSET ERRMSG1
         JMP      WERROR2

SCAN4:
ASSUME   DS:NOTHING
;BX is number of floppies
         OR       BX,BX
         JZ       BADNDRV                 ;User error
         CMP      BX,4
         JA       BADNDRV                 ;User error
         POP      DS
ASSUME   DS:CODE
         PUSH     BX                      ;Save unit count
ABORT:   LDS      BX,[PTRSAV]
ASSUME   DS:NOTHING
         POP      AX
         MOV      BYTE PTR [BX].MEDIA,AL           ;Unit count
         MOV      [DRVMAX],AL
         MOV      WORD PTR [BX].TRANS,OFFSET DRV$INIT ;SET
                                                ;BREAK ADDRESS
         MOV      [BX].TRANS+2,CS
         MOV      WORD PTR [BX].COUNT,OFFSET INITAB
                              ;SET POINTER TO BPB ARRAY
         MOV      [BX].COUNT+2,CS
         JMP      EXIT
;
; PUT SWITCH IN CL, VALUE IN BX
;
SCAN_SWITCH:
```

```
        XOR     BX,BX
        MOV     CX,BX
        LODSB
        CMP     AL,10
        JZ      NUMRET
        CMP     AL,"-"
        JZ      GOT_SWITCH
        CMP     AL,"/"
        JNZ     SCAN_SWITCH
GOT_SWITCH:
        CMP     BYTE PTR [SI+1],":"
        JNZ     TERROR
        LODSB
        OR      AL,20H          ; CONVERT TO LOWERCASE
        MOV     CL,AL           ; GET SWITCH
        LODSB                   ; SKIP ":"
;
;  GET NUMBER POINTED TO BY [SI]
;
;  WIPES OUT AX,DX ONLY     BX RETURNS NUMBER
;
GETNUM1:LODSB
        SUB     AL,"0"
        JB      CHKRET
        CMP     AL,9
        JA      CHKRET
        CBW
        XCHG    AX,BX
        MOV     DX,10
        MUL     DX
        ADD     BX,AX
        JMP     GETNUM1

CHKRET: ADD     AL,"0"
        CMP     AL," "
        JBE     NUMRET
        CMP     AL,"-"
        JZ      NUMRET
        CMP     AL,"/"
        JZ      NUMRET
TERROR:
        POP     DS              ; GET RID OF RETURN ADDRESS
        JMP     WERROR
NUMRET: DEC     SI
        RET

ERRMSG1 DB      "SMLDRV: Bad number of drives",13,10,"$"
ERRMSG2 DB      "SMLDRV: Invalid parameter",13,10,"$"
CODE    ENDS
        END
```

2.12.2 Character Device Driver

The following program illustrates a character device driver program.

```
;******************** A CHARACTER DEVICE ******************

TITLE   VT52 CONSOLE FOR 2.0    (IBM)

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;       IBM ADDRESSES FOR I/O
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

        CR=13                   ;CARRIAGE RETURN
        BACKSP=8                ;BACKSPACE
        ESC=1BH
        BRKADR=6CH              ;006C  BREAK VECTOR ADDRESS
        ASNMAX=200              ;SIZE OF KEY ASSIGNMENT BUFFER

CODE    SEGMENT BYTE

   ASSUME CS:CODE,DS:NOTHING,ES:NOTHING
;-----------------------------------------------------------
;
;       C O N - CONSOLE DEVICE DRIVER
;
CONDEV:                                 ;HEADER FOR DEVICE "CON"
        DW      -1,-1
        DW      1000000000010011B       ;CON IN AND CON OUT
        DW      STRATEGY
        DW      ENTRY
        DB      'CON     '

;------------------------------------------------------------
;
;       COMMAND JUMP TABLES
CONTBL:
        DW      CON$INIT
        DW      EXIT
        DW      EXIT
        DW      CMDERR
        DW      CON$READ
        DW      CON$RDND
        DW      EXIT
        DW      CON$FLSH
        DW      CON$WRIT
        DW      CON$WRIT
        DW      EXIT
        DW      EXIT

CMDTABL DB      'A'
```

```
        DW      CUU                 ;cursor up
        DB      'B'
        DW      CUD                 ;cursor down
        DB      'C'
        DW      CUF                 ;cursor forward
        DB      'D'
        DW      CUB                 ;cursor back
        DB      'H'
        DW      CUH                 ;cursor position
        DB      'J'
        DW      ED                  ;erase display
        DB      'K'
        DW      EL                  ;erase line
        DB      'Y'
        DW      CUP                 ;cursor position
        DB      'j'
        DW      PSCP                ;save cursor position
        DB      'k'
        DW      PRCP                ;restore cursor position
        DB      'y'
        DW      RM                  ;reset mode
        DB      'x'
        DW      SM                  ;set mode
        DB      00

PAGE
;-----------------------------------------------------
;
;       Device entry point
;
CMDLEN  =       0       ;LENGTH OF THIS COMMAND
UNIT    =       1       ;SUB UNIT SPECIFIER
CMD     =       2       ;COMMAND CODE
STATUS  =       3       ;STATUS
MEDIA   =       13      ;MEDIA DESCRIPTOR
TRANS   =       14      ;TRANSFER ADDRESS
COUNT   =       18      ;COUNT OF BLOCKS OR CHARACTERS
START   =       20      ;FIRST BLOCK TO TRANSFER

PTRSAV  DD      0

STRATP  PROC    FAR

STRATEGY:
        MOV     WORD PTR CS:[PTRSAV],BX
        MOV     WORD PTR CS:[PTRSAV+2],ES
        RET

STRATP  ENDP

ENTRY:
        PUSH    SI
        PUSH    AX
        PUSH    CX
        PUSH    DX
```

```
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        PUSH    BX

        LDS     BX,CS:[PTRSAV]   ;GET POINTER TO I/O PACKET

        MOV     CX,WORD PTR DS:[BX].COUNT     ;CX = COUNT

        MOV     AL,BYTE PTR DS:[BX].CMD
        CBW
        MOV     SI,OFFSET CONTBL
        ADD     SI,AX
        ADD     SI,AX
        CMP     AL,11
        JA      CMDERR

        LES     DI,DWORD PTR DS:[BX].TRANS

        PUSH    CS
        POP     DS

        ASSUME  DS:CODE

        JMP     WORD PTR [SI]                 ;GO DO COMMAND

PAGE
;=====================================================
;=
;=      SUBROUTINES SHARED BY MULTIPLE DEVICES
;=
;=====================================================
;-----------------------------------------------------
;
;       EXIT - ALL ROUTINES RETURN THROUGH THIS PATH
;
BUS$EXIT:                                ;DEVICE BUSY EXIT
        MOV     AH,00000011B
        JMP     SHORT ERR1

CMDERR:
        MOV     AL,3                 ;UNKNOWN COMMAND ERROR

ERR$EXIT:
        MOV     AH,10000001B             ;MARK ERROR RETURN
        JMP     SHORT ERR1

EXITP   PROC    FAR

EXIT:   MOV     AH,00000001B
ERR1:   LDS     BX,CS:[PTRSAV]
        MOV     WORD PTR [BX].STATUS,AX ;MARK
                                         ;OPERATION COMPLETE
```

```
        POP     BX
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     DX
        POP     CX
        POP     AX
        POP     SI
        RET                             ;RESTORE REGS AND RETURN
EXITP   ENDP
;------------------------------------------------
;
;       BREAK KEY HANDLING
;
BREAK:
        MOV     CS:ALTAH,3              ;INDICATE BREAK KEY SET
INTRET: IRET

PAGE
;
;       WARNING - Variables are very order dependent,
;                 so be careful when adding new ones!
;
WRAP    DB      0                    ; 0 = WRAP, 1 = NO WRAP
STATE   DW      S1
MODE    DB      3
MAXCOL  DB      79
COL     DB      0
ROW     DB      0
SAVCR   DW      0
ALTAH   DB      0                    ;Special key handling

;-------------------------------------------------------
;
;    CHROUT - WRITE OUT CHAR IN AL USING CURRENT ATTRIBUTE
;
ATTRW   LABEL   WORD
ATTR    DB      00000111B            ;CHARACTER ATTRIBUTE
BPAGE   DB      0                    ;BASE PAGE
base    dw      0b800h

chrout: cmp     al,13
        jnz     trylf
        mov     [col],0
        jmp     short setit

trylf:  cmp     al,10
        jz      lf
        cmp     al,7
        jnz     tryback
torom:
        mov     bx,[attrw]
        and     bl,7
        mov     ah,14
```

```
        int     10h
ret5:   ret

tryback:
        cmp     al,8
        jnz     outchr
        cmp     [col],0
        jz      ret5
        dec     [col]
        jmp     short setit

outchr:
        mov     bx,[attrw]
        mov     cx,1
        mov     ah,9
        int     10h
        inc     [col]
        mov     al,[col]
        cmp     al,[maxcol]
        jbe     setit
        cmp     [wrap],0
        jz      outchrl
        dec     [col]
        ret
outchrl:
        mov     [col],0
lf:     inc     [row]
        cmp     [row],24
        jb      setit
        mov     [row],23
        call    scroll

setit:  mov     dh,row
        mov     dl,col
        xor     bh,bh
        mov     ah,2
        int     10h
        ret

scroll: call    getmod
        cmp     al,2
        jz      myscroll
        cmp     al,3
        jz      myscroll
        mov     al,10
        jmp     torom
myscroll:
        mov     bh,[attr]
        mov     bl,' '
        mov     bp,80
        mov     ax,[base]
        mov     es,ax
        mov     ds,ax
        xor     di,di
        mov     si,160
```

```
        mov     cx,23*80
        cld
        cmp     ax,0b800h
        jz      colorcard

        rep     movsw
        mov     ax,bx
        mov     cx,bp
        rep     stosw
sret:   push    cs
        pop     ds
        ret

colorcard:
        mov     dx,3dah
wait2:  in      al,dx
        test    al,8
        jz      wait2
        mov     al,25h
        mov     dx,3d8h
        out     dx,al           ;turn off video
        rep     movsw
        mov     ax,bx
        mov     cx,bp
        rep     stosw
        mov     al,29h
        mov     dx,3d8h
        out     dx,al           ;turn on video
        jmp     sret

GETMOD: MOV     AH,15
        INT     16             ;get column information
        MOV     BPAGE,BH
        DEC     AH
        MOV     WORD PTR MODE,AX
        RET
;-------------------------------------------------------
;
;       CONSOLE READ ROUTINE
;
CON$READ:
        JCXZ    CON$EXIT
CON$LOOP:
        PUSH    CX              ;SAVE COUNT
        CALL    CHRIN           ;GET CHAR IN AL
        POP     CX
        STOSB                   ;STORE CHAR AT ES:DI
        LOOP    CON$LOOP
CON$EXIT:
        JMP     EXIT
;----------------------------------------------------------
;
;       INPUT SINGLE CHAR INTO AL
;
CHRIN:  XOR     AX,AX
```

```
        XCHG    AL,ALTAH        ;GET CHARACTER & ZERO ALTAH
        OR      AL,AL
        JNZ     KEYRET

INAGN:  XOR     AH,AH
        INT     22
ALT10:
        OR      AX,AX          ;Check for non-key after BREAK
        JZ      INAGN
        OR      AL,AL          ;SPECIAL CASE?
        JNZ     KEYRET
        MOV     ALTAH,AH            ;STORE SPECIAL KEY
KEYRET: RET
;-------------------------------------------------------------
;
;       KEYBOARD NON DESTRUCTIVE READ, NO WAIT
;
CON$RDND:
        MOV     AL,[ALTAH]
        OR      AL,AL
        JNZ     RDEXIT

RD1:    MOV     AH,1
        INT     22
        JZ      CONBUS
        OR      AX,AX
        JNZ     RDEXIT
        MOV     AH,0
        INT     22
        JMP     CON$RDND

RDEXIT: LDS     BX,[PTRSAV]
        MOV     [BX].MEDIA,AL
EXVEC:  JMP     EXIT
CONBUS: JMP     BUS$EXIT
;-------------------------------------------------------------
;
;       KEYBOARD FLUSH ROUTINE
;
CON$FLSH:
        MOV     [ALTAH],0           ;Clear out holding buffer

        PUSH    DS
        XOR     BP,BP
        MOV     DS,BP                     ;Select segment 0
        MOV     DS:BYTE PTR 41AH,1EH      ;Reset KB queue head
                                          ;pointer
        MOV     DS:BYTE PTR 41CH,1EH      ;Reset tail pointer
        POP     DS
        JMP     EXVEC
;-------------------------------------------------------------
;
;       CONSOLE WRITE ROUTINE
;
CON$WRIT:
```

```
        JCXZ    EXVEC
        PUSH    CX
        MOV     AH,3                ;SET CURRENT CURSOR POSITION
        XOR     BX,BX
        INT     16
        MOV     WORD PTR [COL],DX
        POP     CX

CON$LP: MOV     AL,ES:[DI]          ;GET CHAR
        INC     DI
        CALL    OUTC                ;OUTPUT CHAR
        LOOP    CON$LP              ;REPEAT UNTIL ALL THROUGH
        JMP     EXVEC

COUT:   STI
        PUSH    DS
        PUSH    CS
        POP     DS
        CALL    OUTC
        POP     DS
        IRET

OUTC:   PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    ES
        PUSH    BP
        CALL    VIDEO
        POP     BP
        POP     ES
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET


;----------------------------------------------------------
;
;       OUTPUT SINGLE CHAR IN AL TO VIDEO DEVICE
;
VIDEO:  MOV     SI,OFFSET STATE
        JMP     [SI]

S1:     CMP     AL,ESC                      ;ESCAPE SEQUENCE?
        JNZ     S1B
        MOV     WORD PTR [SI],OFFSET S2
        RET

S1B:    CALL    CHROUT
S1A:    MOV     WORD PTR [STATE],OFFSET S1
        RET
```

```
S2:     PUSH    AX
        CALL    GETMOD
        POP     AX
        MOV     BX,OFFSET CMDTABL-3
S7A:    ADD     BX,3
        CMP     BYTE PTR [BX],0
        JZ      S1A
        CMP     BYTE PTR [BX],AL
        JNZ     S7A
        JMP     WORD PTR [BX+1]


MOVCUR: CMP     BYTE PTR [BX],AH
        JZ      SETCUR
        ADD     BYTE PTR [BX],AL
SETCUR: MOV     DX,WORD PTR COL
        XOR     BX,BX
        MOV     AH,2
        INT     16
        JMP     S1A


CUP:    MOV     WORD PTR [SI],OFFSET CUP1
        RET
CUP1:   SUB     AL,32
        MOV     BYTE PTR [ROW],AL
        MOV     WORD PTR [SI],OFFSET CUP2
        RET
CUP2:   SUB     AL,32
        MOV     BYTE PTR [COL],AL
        JMP     SETCUR

SM:     MOV     WORD PTR [SI],OFFSET S1A
        RET


CUH:    MOV     WORD PTR COL,0
        JMP     SETCUR

CUF:    MOV     AH,MAXCOL
        MOV     AL,1
CUF1:   MOV     BX,OFFSET COL
        JMP     MOVCUR

CUB:    MOV     AX,00FFH
        JMP     CUF1

CUU:    MOV     AX,00FFH
CUU1:   MOV     BX,OFFSET ROW
        JMP     MOVCUR

CUD:    MOV     AX,23*256+1
        JMP     CUU1
```

```
PSCP:   MOV     AX,WORD PTR COL
        MOV     SAVCR,AX
        JMP     SETCUR

PRCP:   MOV     AX,SAVCR
        MOV     WORD PTR COL,AX
        JMP     SETCUR

ED:     CMP     BYTE PTR [ROW],24
        JAE     EL1

        MOV     CX,WORD PTR COL
        MOV     DH,24
        JMP     ERASE

EL1:    MOV     BYTE PTR [COL],0
EL:     MOV     CX,WORD PTR [COL]
EL2:    MOV     DH,CH
ERASE:  MOV     DL,MAXCOL
        MOV     BH,ATTR
        MOV     AX,0600H
        INT     16
ED3:    JMP     SETCUR


RM:     MOV     WORD PTR [SI],OFFSET RM1
        RET
RM1:    XOR     CX,CX
        MOV     CH,24
        JMP     EL2

CON$INIT:
        int     11h
        and     al,00110000b
        cmp     al,00110000b
        jnz     iscolor
        mov     [base],0b000h           ;look for bw card
iscolor:
        cmp     al,00010000b            ;look for 40 col mode
        ja      setbrk
        mov     [mode],0
        mov     [maxcol],39

setbrk:
        XOR     BX,BX
        MOV     DS,BX
        MOV     BX,BRKADR
        MOV     WORD PTR [BX],OFFSET BREAK
        MOV     WORD PTR [BX+2],CS

        MOV     BX,29H*4
        MOV     WORD PTR [BX],OFFSET COUT
        MOV     WORD PTR [BX+2],CS
```

```
        LDS     BX,CS:[PTRSAV]
        MOV     WORD PTR [BX].TRANS,OFFSET CON$INIT
                              ;SET BREAK ADDRESS
        MOV     [BX].TRANS+2,CS
        JMP     EXIT

CODE    ENDS
        END
```

Chapter 3
MS-DOS Technical Information

CHAPTER 3

MS-DOS TECHNICAL INFORMATION

3.1 MS-DOS INITIALIZATION

MS-DOS initialization consists of several steps. Typically, a ROM (Read Only Memory) bootstrap obtains control, and then reads the boot sector off the disk. The boot sector then reads the following files:

IO.SYS
MSDOS.SYS

Once these files are read, the boot process begins.

3.2 THE COMMAND PROCESSOR

The command processor supplied with MS-DOS (file COMMAND.COM.) consists of three parts:

1. A resident part resides in memory immediately following MSDOS.SYS and its data area. This part contains routines to process Interrupts 23H (Control-C Exit Address) and 24H (Critical Error Handler Address), as well as a routine to reload the transient part, if needed. All standard MS-DOS error handling is done within this part of COMMAND.COM. This includes displaying error messages and processing the Abort, Retry, or Ignore messages.

2. An initialization part follows the resident part. During startup, the initialization part is given control; it contains the AUTOEXEC file processor setup routine. The initialization part determines the segment address at which programs can be loaded. It is overlaid by the first program COMMAND.COM loads because it is no longer needed.

3. A transient part is loaded at the high end of memory. This part contains all of the internal command processors and the batch file processor.

   The transient part of the command processor produces the system prompt (such as A>), reads the command from keyboard (or batch file), and causes it to be executed. For external commands, this part builds a command line and issues the EXEC system call (Function Request 4B00H) to load and transfer control to the program.

3.3 MS-DOS DISK ALLOCATION

The MS-DOS area is formatted as follows:

Reserved area - variable size

First copy of file allocation table - variable size

Additional copies of file allocation table - variable size (optional)

Root directory - variable size

File data area

Space for a file in the data area is not pre-allocated. The space is allocated one cluster at a time. A cluster consists of one or more consecutive sectors (the number of sectors in a cluster must be a power of 2); The cluster size is determined at format time. All of the clusters for a file are "chained" together in the File Allocation Table (FAT). (Refer to Section 3.5, "File Allocation Table," for more information on the FAT.) A second copy of the FAT is normally kept for consistency except in the case of extremely reliable storage such as a virtual RAM disk. Should the disk develop a bad sector in the middle of the first FAT, the second can be used. This avoids loss of data due to an unreadable FAT.

3.4 MS-DOS DISK DIRECTORY

FORMAT builds the root directory for all disks. Its location on disk and the maximum number of entries are dependent on the media.

Since directories other than the root directory are regarded as files by MS-DOS, there is no limit to the number of files they may contain.

All directory entries are 32 bytes in length, and are in the following format (note that byte offsets are in hexadecimal):

0-7 Filename. Eight characters, left aligned and padded, if necessary, with blanks. The first byte of this field indicates the file status as follows:

00H The directory entry has never been used. This is used to limit the length of directory searches, for performance reasons.

05H Indicates that the first character of the filename actually has an E5H character.

2EH The entry is for a directory. If the second byte is also 2EH, then the cluster field contains the cluster number of this directory's parent directory (0000H if the parent directory is the root directory). Otherwise, bytes 01H through 0AH are all spaces, and the cluster field contains the cluster number of this directory.

E5H The file was used, but it has been erased.

Any other character is the first character of a filename.

8-0A Filename extension.

0B File attribute. The attribute byte is mapped as follows (values are in hexadecimal):

01 File is marked read-only. An attempt to open the file for writing using the Open Handle system call (Function Request 3DH) results in an error code being returned. This value can be used along with other values below. Attempts to delete the file with the Delete File system call (13H) or Delete

Directory Entry (41H) will also fail.

02 Hidden file. The file is excluded from normal directory searches.

04 System file. The file is excluded from normal directory searches.

08 The entry contains the volume label in the first 11 bytes. The entry contains no other usable information (except date and time of creation), and may exist only in the root directory.

10 The entry defines a subdirectory, and is excluded from normal directory searches.

20 Archive bit. The bit is set to "on" whenever the file has been written to and closed.

Note: The system files (IO.SYS and MSDOS.SYS) are marked as read-only, hidden, and system files. Files can be marked hidden when they are created. Also, the read-only, hidden, system, and archive attributes may be changed through the Get/Set File Attributes system call (Function Request 43H).

0C-15 RESERVED.

16-17 Time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows (bit 7 on left, 0 on right):

Offset 17H

H	H	H	H	H	M	M	M

Offset 16H

M	M	M	S	S	S	S	S

where:

H is the binary number of hours (0-23)
M is the binary number of minutes (0-59)
S is the binary number of two-second increments

18-19 Date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

Offset 19H

| Y | Y | Y | Y | Y | Y | Y | M |

Offset 18H

| M | M | M | D | D | D | D | D |

where:

Y is 0-119 (1980-2099)
M is 1-12
D is 1-31

1A-1B Starting cluster; the cluster number of the first cluster in the file.

Note that the first cluster for data space on all disks is cluster 002.

The cluster number is stored with the least significant byte first.

Note

Refer to Sections 3.5.1 and 3.5.2 for details about converting cluster numbers to logical sector numbers.

1C-1F File size in bytes. The first word of this four-byte field is the low-order part of the size.

3.5 FILE ALLOCATION TABLE (FAT)

The following information is included for system programmers who wish to write installable device drivers. This section explains how MS-DOS uses the File Allocation Table to convert the clusters of a file to logical sector numbers to allocate disk space for a file. The driver is then responsible for locating the logical sector on disk. Programs should use the MS-DOS file management function calls for accessing files; programs that access the FAT are not guaranteed to be upwardly-compatible with future releases of MS-DOS.

The File Allocation Table is an array of 12-bit entries (1.5 bytes) for each cluster on the disk. For disks containing more than 4085 (note that 4085 is the correct number) clusters, a 16-bit FAT entry is used.

The first byte may be used by the device driver as a FAT ID byte for media determination. The first two FAT entries are reserved.

The third FAT entry, which starts at byte offset 4, begins the mapping of the data area (cluster 002). Files in the data area are not always written sequentially on the disk. The data area is allocated one cluster at a time, skipping over clusters already allocated. The first free cluster following the last cluster allocated for that file will be the next cluster allocated, regardless of its physical location on the disk. This permits the most efficient utilization of disk space because clusters made available by erasing files can be allocated for new files.

Each FAT entry contains three or four hexadecimal characters depending on whether it is a 12- or 16-bit entry:

(0)000 If the cluster is unused and available.

(F)FF7 The cluster has a bad sector in it if this cluster is not part of any cluster chain. MS-DOS will not allocate such a cluster. Chkdsk counts the number of bad clusters for its report. These bad clusters are not part of any allocation chain.

(F)FF8-FFF Indicates the last cluster of a file.

(X)XXX Any other characters that are the cluster number of the next cluster in the file. The cluster number of the first cluster in the file is kept in the file's directory entry.

The File Allocation Table always begins on the first sector after the reserved sectors. If the FAT is larger than one sector, the sectors are contiguous. Two copies of the FAT are usually written for data integrity. The FAT is read into one of the MS-DOS buffers whenever needed (open, read, write, etc.). For performance reasons, this buffer is given a high priority to keep it in memory as long as possible.

3.5.1 How To Use the FAT (12-bit FAT Entries)

Use the directory entry to find the starting cluster of the file. Next, to locate each subsequent cluster of the file:

1. Multiply the cluster number just used by 1.5 (each FAT entry is 1.5 bytes long).

2. The whole part of the product is an offset into the FAT, pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.

3. Use a MOV instruction to move the word at the calculated FAT offset into a register.

4. If the last cluster used was an even number, keep the low-order 12 bits of the register by ANDing it with FFF; otherwise, keep the high-order 12 bits by shifting the register right 4 bits with a SHR instruction.

5. If the resultant 12 bits are FF8H-FFFH, the file contains no more clusters. Otherwise, the 12 bits contain the cluster number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by DEBUG):

1. Subtract 2 from the cluster number.

2. Multiply the result by the number of sectors per cluster.

3. Add to this result the logical sector number of the beginning of the data area.

3.5.2 How To Use The FAT (16-bit FAT Entries)

Use the directory entry to get the starting cluster of the file. To find the next file cluster:

1. Multiply the cluster number used by 2 (each FAT entry is 2 bytes).

2. Use a MOV WORD instruction to move the word at the calculated FAT offset into a register.

3. If the resultant 16 bits are FFF8-FFFFH, then there are no more clusters in the file. Otherwise, the 16 bits contain the cluster number of the next cluster at the file.

3.6 MS-DOS STANDARD DISK FORMATS

On an MS-DOS disk, it is recommended that the clusters be arranged on disk to minimize head movement for multi-sided media. All of the space on a track (or cylinder) is allocated before moving on to the next track. This is accomplished by using the sequential sectors on the lowest-numbered head, then all the sectors on the next head, and so on, until all sectors on all heads of the track are used. The next sector to be used will be sector 1 on head 0 of the next track.

The formats in Table 3.1 are considered to be standard and should be readable if at all possible.

Table 3.1 MS-DOS Standard Disk Formats

Disk Size (in inches)	3-1/2 or 5-1/4				5-1/4					8		
Number of tracks	80	80	80	80	40	40	40	40	80	77	77	77
3 byte JUMP												
8 byte name												
WORD bytes/sector	00	00	00	00	00	00	00	00	00	80	80	00
	02	02	02	02	02	02	02	02	02	00	00	04
BYTE cluster size	02	02	02	02	01	02	01	02	01	04	04	01
WORD reserved sectors	01	01	01	01	01	01	01	01	01	01	04	01
	00	00	00	00	00	00	00	00	00	00	00	00
BYTE # FATs	02	02	02	02	02	02	02	02	02	02	02	02
WORD # Dir entries	70	70	70	70	40	70	40	70	E0	44	44	C0
	00	00	00	00	00	00	00	00	00	00	00	00
WORD # sectors	D0	A0	80	00	68	D0	40	80	60	D2	D2	68
	02	05	02	05	01	02	01	02	09	07	07	02
BYTE media	F8	F9	FA	FB	FC	FD	FE	FF	F9	FE	FD	FE
WORD sectors/FAT	02	03	01	02	02	02	01	01	07	06	06	02
	00	00	00	00	00	00	00	00	00	00	00	00
WORD sectors/track	09	09	08	08	09	09	08	08	0F	1A	1A	08
	00	00	00	00	00	00	00	00	00	00	00	00
WORD # heads	01	02	01	02	01	02	01	02	02	01	01	02
	00	00	00	00	00	00	00	00	00	00	00	00
WORD hidden sectors	00	00	00	00	00	00	00	00	00	00	00	00
	00	00	00	00	00	00	00	00	00	00	00	00

Chapter 4
MS-DOS Control Blocks and Work Areas

CHAPTER 4

MS-DOS CONTROL BLOCKS AND WORK AREAS

4.1 TYPICAL MS-DOS MEMORY MAP

Interrupt vector table

Optional extra space (used by IBM for ROM data area)

IO.SYS - MS-DOS interface to hardware

MSDOS.SYS - MS-DOS interrupt handlers, service routines (Interrupt 21H functions)

MS-DOS buffers, control areas, and installed device drivers

Resident part of COMMAND.COM - Interrupt handlers for Interrupts 22H (Terminate Process Exit Address), 23H (Control-C Handler Address), 24H (Critical Error Handler Address) and code to reload the transient part

External command or utility - (.COM or .EXE file)

User stack for .COM files (256 bytes)

Transient part of COMMAND.COM - Command interpreter, internal commands, batch processor

User memory is allocated from the lowest end of available memory that will meet the allocation request.

4.2 MS-DOS PROGRAM SEGMENT

When an external command is typed, or when you execute a program through the EXEC system call, MS-DOS determines the lowest available free memory address to use as the start of the program. This area is called the Program Segment.

The first 256 bytes of the Program Segment are set up by the EXEC system call for the program being loaded into memory. The program is then loaded following this block. An .EXE file with minalloc and maxalloc both set to zero is loaded as high as possible.

At offset 0 within the Program Segment, MS-DOS builds the Program Segment Prefix control block. The program returns from EXEC by one of five methods:

1. By issuing an Interrupt 21H with AH=4CH
2. By issuing an Interrupt 21H with AH=31H (Keep Process)
3. A long jump to offset 0 in the Program Segment Prefix
4. By issuing an Interrupt 20H with CS:0 pointing at the PSP
5. By issuing an Interrupt 21H with register AH=0 and with CS:0 pointing at the PSP.

> **Note**
>
> Methods 1 and 2 are preferred for both functionality and best operation in future versions of MS-DOS.

All five methods result in transferring control to the program that issued the EXEC. Using method 1 or 2 allows a completion code to be returned. During this returning process, Interrupts 22H, 23H, and 24H (Terminate Process Exit Address, Control-C Handler Address, and Critical Error Handler Address) addresses are restored from the values saved in the Program Segment Prefix of the terminating program. Control is then given to the terminate address. If this is a program returning to COMMAND.COM, control transfers to its resident portion. If a batch file was in process, it is continued; otherwise, COMMAND.COM performs a checksum on the transient part, reloads it if necessary, then issues the system prompt and waits for you to type the next command.

When a program receives control, the following conditions are in effect:

For all programs:

The segment address of the passed environment is contained at offset 2CH in the Program Segment Prefix.

The environment is a series of ASCII strings (totaling less than 32K) in the form:

NAME=parameter

Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros.

Following the last byte of zeros is a set of initial arguments passed to a program that contains a word count followed by an ASCIZ string. If the file is found in the current directory, the ASCIZ string contains the drive and pathname of the executable program as passed to the EXEC function call. If the file is found in the path, the filename is concatenated with the information in the path. Programs may use this area to determine where the program was loaded.

The environment built by the command processor contains at least a COMSPEC= string (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last Path and Prompt commands issued will also be in the environment, along with any environment strings defined with the MS-DOS Set command.

The environment that is passed is a copy of the invoking process environment. If your application uses a "keep process" concept, you should be aware that the copy of the environment passed to you is static. That is, it will not change even if subsequent Set, Path, or Prompt commands are issued. Conversely, any modification of the passed environment by the application will not be reflected in the parent process environment. For instance, a program cannot change the MS-DOS environment values as the Set command does.

The Disk Transfer Address (DTA) is set to 80H (default DTA in the Program Segment Prefix). At 5CH and 6CH in the Program Segment Prefix are file control blocks. These are formatted from the first two parameters, typed when the command was entered.

If either parameter contained a pathname, then the corresponding FCB contains only the valid drive number. The filename field will not be valid.

An unformatted parameter area at 81H contains all the characters typed after the command (including leading and imbedded delimiters), with the byte at 80H set to the number of characters. If the <, >, or parameters were typed on the command line, they (and the filenames associated with them) will not appear in this area; redirection of standard input and output is transparent to applications.

Offset 6 (one word) contains the number of bytes available in the segment.

Register AX indicates whether or not the drive specifiers (entered with the first two parameters) are valid, as follows:

> AL=FF if the first parameter contained an invalid drive specifier (otherwise AL=00)
>
> AH=FF if the second parameter contained an invalid drive specifier (otherwise AH=00)

Offset 2 (one word) contains the segment address of the first byte of <u>unavailable</u> memory. Programs must not modify addresses beyond this point unless they were obtained by allocating memory via the Allocate Memory system call (Function Request 48H).

<u>For Executable (.EXE) programs</u>:

DS and ES registers are set to point to the Program Segment Prefix.

CS,IP,SS, and SP registers are set to the values set by MS-LINK in the .EXE image.

<u>For Executable (.COM) programs</u>:

All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.

All of user memory is allocated to the program. If the program invokes another program through Function Request 4BH, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being executed.

The Instruction Pointer (IP) is set to 100H.

The Stack Pointer register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.

A word of zeros is placed on top of the stack. This is to allow a user program to exit to COMMAND.COM by doing a RET instruction last. This assumes, however, that the user has maintained his stack and code segments.

Figure 4.1 illustrates the format of the Program Segment Prefix. All offsets are in hexadecimal.

(Offsets in Hex)

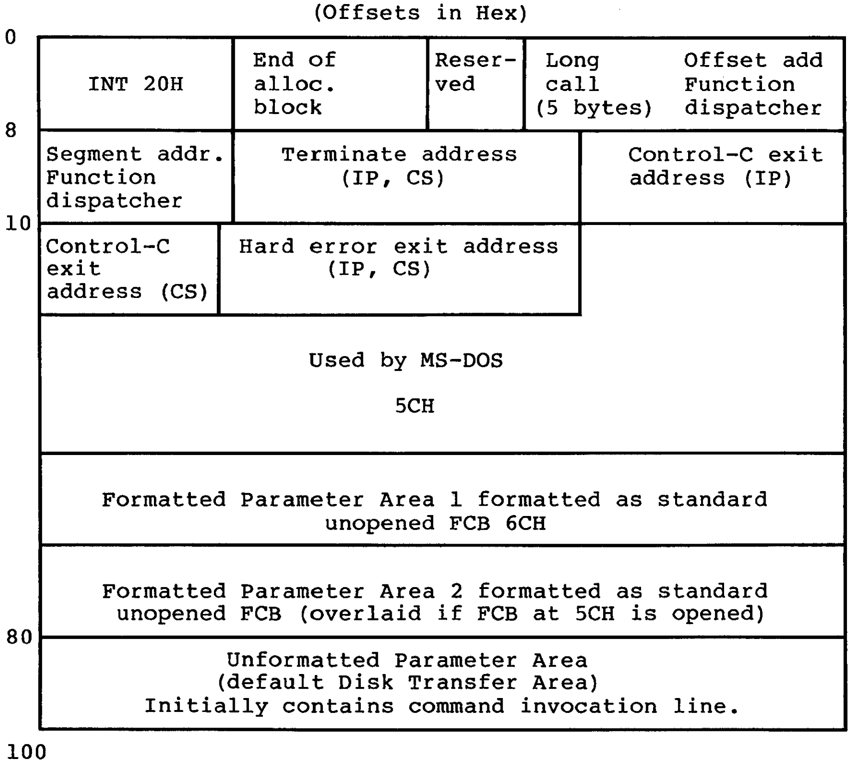


Figure 4.1 Program Segment Prefix

Important

Programs must not alter any part of the Program Segment Prefix below offset 5CH.

Chapter 5
.EXE File Structure and Loading

CHAPTER 5

.EXE FILE STRUCTURE AND LOADING

Note

This chapter describes .EXE file structure and loading procedures for systems that use a version of MS-DOS that is lower than 2.0. For MS-DOS 2.0 and higher, use Function Request 4B00H, Load and Execute a Program, to load (or load and execute) an .EXE file.

The .EXE files produced by the Microsoft(R) Linker (MS-LINK) consist of two parts:

Control and relocation information

The load module

The control and relocation information is at the beginning of the file in an area called the header. The load module immediately follows the header.

The header is formatted as follows. (Note that offsets are in hexadecimal.)

Offset	Contents
00-01	Must contain 4DH, 5AH.
02-03	Number of bytes contained in last page; this is useful in reading overlays.
04-05	Size of the file in 512-byte pages, including the header.
06-07	Number of relocation entries in table.
08-09	Size of the header in 16-byte paragraphs.

	This is used to locate the beginning of the load module in the file.
0A-0B	Minimum number of 16-byte paragraphs required above the end of the loaded program.
0C-0D	Maximum number of 16-byte paragraphs required above the end of the loaded program. If both minalloc and max-alloc are 0, then the program will be loaded as high as possible.
0E-0F	Initial value to be loaded into stack segment before starting program execution. This must be adjusted by relocation.
10-11	Value to be loaded into the SP register before starting program execution.
12-13	Negative sum of all the words in the file.
14-15	Initial value to be loaded into the IP register before starting program execution.
16-17	Initial value to be loaded into the CS register before starting program execution. This must be adjusted by relocation.
18-19	Relative byte offset from beginning of run file to relocation table.
1A-1B	The number of the overlay as generated by MS-LINK.

The relocation table follows the formatted area described above. This table consists of a variable number of relocation items. Each relocation item contains two fields: a two-byte offset value, followed by a two-byte segment value. These two fields contain the offset into the load module of a word which requires modification before the module is given control. The following steps describe this process:

1. The formatted part of the header is read into memory. Its size is 1BH.

2. A portion of memory is allocated depending on the size of the load module and the allocation numbers (0A-0B and 0C-0D). MS-DOS attempts to allocate FFFFH paragraphs. This will always fail, returning the size of the largest free block. If this block is smaller than minalloc and loadsize, then there will be no memory error. If this block is larger than maxalloc and loadsize, MS-DOS will allocate (maxalloc + loadsize). Otherwise, MS-DOS will allocate the largest free block of memory.

3. A Program Segment Prefix is built in the lowest part of the allocated memory.

4. The load module size is calculated by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted based on the contents of offsets 02-03. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.

5. The load module is read into memory beginning with the start segment.

6. The relocation table items are read into a work area.

7. Each relocation table item segment value is added to the start segment value. This calculated segment, plus the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.

8. Once all relocation items have been processed, the SS and SP registers are set from the values in the header. Then, the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is the initial CS:IP to transfer to before starting execution of the program.

Chapter 6
Intel Relocatable Object Module Formats

CHAPTER 6

INTEL RELOCATABLE OBJECT MODULE FORMATS

6.1 INTRODUCTION

This chapter presents the object record formats that define the relocatable object language for the 8086 microprocessor. The 8086 object language is the output of all language translators that have the 8086 as the target processor and are to be linked using the Microsoft Linker. The 8086 object language is input and output for object language processors such as linkers and librarians.

The 8086 object module formats permit you to specify relocatable memory images that may be linked together. Capabilities are provided that allow efficient use of the memory mapping facilities of the 8086 microprocessor.

The following table lists the record formats that are supported by Microsoft. These record formats are described in this chapter. Record formats that are preceded by an asterisk (*) <u>deviate from the Intel(R) specification</u>.

Table 6.1 Object Module Record Formats

```
   T-MODULE HEADER RECORD
   LIST OF NAMES RECORD
  *SEGMENT DEFINITION RECORD
  *GROUP DEFINITION RECORD
  *TYPE DEFINITION RECORD

 Symbol Definition Records
  *PUBLIC NAMES DEFINITION RECORD
  *EXTERNAL NAMES DEFINITION RECORD
  *LINE NUMBERS RECORD

 Data Records
   LOGICAL ENUMERATED DATA RECORD
   LOGICAL ITERATED DATA RECORD

   FIXUP RECORD
  *MODULE END RECORD
   COMMENT RECORD
```

6.2 DEFINITION OF TERMS

The following terms are fundamental to the 8086 relocation and linkage.

OMF - Object Module Formats.

MAS - Memory Address Space. The 8086 MAS is 1 megabyte (1,048,576). Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

MODULE - an "inseparable" collection of object code and other information produced by a translator.

T-MODULE - A module created by a translator, such as Pascal or FORTRAN.

The following restrictions apply to object modules:

1. Every module should have a name. Translators will provide names for T-modules, providing a default name (possibly the filename or a null name) if neither source code nor user specifies otherwise.

2. Every T-module in a collection of linked modules must have a different name, so that symbolic debugging systems can distinguish the various line numbers and local symbols. This restriction is not required by the Linker, and is not enforced by it.

FRAME - A contiguous region of 64K of MAS, beginning on a paragraph boundary (i.e., on a multiple of 16 bytes). This concept is useful because the content of the four 8086 segment registers defines four (possibly overlapping) FRAMEs; no 16-bit address in the 8086 code can access a memory location outside of the current four FRAMEs.

LSEG - Logical Segment - A contiguous region of memory whose contents are determined at translation time (except for address-binding). Neither size nor location in MAS are necessarily determined at translation time: size, although partially fixed, may not be final because the LSEG may be combined at LINK time with other LSEGs, forming a single LSEG. An LSEG must not be larger than 64K, so that it can fit in a FRAME. This means that any byte in an LSEG may be addressed by a 16-bit offset from the base of a FRAME covering the LSEG.

PSEG - Physical Segment - This term is equivalent to FRAME. Some people prefer "PSEG" to "FRAME" because the terms "PSEG" and "LSEG" reflect the "physical" and "logical" nature of the underlying segments.

FRAME NUMBER - Every FRAME begins on a paragraph boundary. The "paragraphs" in MAS can be numbered from 0 through 65535. These numbers, each of which defines a FRAME, are called FRAME NUMBERS.

PARAGRAPH NUMBER - This term is equivalent to FRAME NUMBER.

PSEG NUMBER - This term is equivalent to FRAME NUMBER.

GROUP - A collection of LSEGs defined at translation time, whose final locations in MAS have been constrained such that there will be at least one FRAME that covers (contains) every LSEG in the collection.

The notation "Gr A(X,Y,Z,)" means that LSEGs X, Y and Z form a group whose name is A. The fact that X, Y and Z are all LSEGs in the same group <u>does not imply any ordering of X, Y and Z in MAS, nor does it imply any contiguity between X, Y and Z</u>.

The Microsoft Linker does not currently allow an LSEG to be a member of more than one group. The Linker will ignore all attempts to place an LSEG in more than one group.

CANONIC - Any location in MAS is contained in exactly 4096 distinct FRAMEs; but one of these FRAMEs can be distinguished because it has a higher FRAME NUMBER. This distinguished FRAME is called the canonic FRAME of the location. In other words, the canonic frame of a given byte is the frame so chosen that the byte's offset from that frame lies in the range 0 to 15 (decimal). Thus, if FOO is a symbol defining a memory location, one may speak of the "canonic FRAME of FOO", or of "FOO's canonic FRAME". By extension, if S is any set of memory locations, then there exists a unique FRAME which has the lowest FRAME NUMBER in the set of canonic FRAMEs of the locations in S. This unique FRAME is called the canonic FRAME of the set S. Thus, we may speak of the canonic FRAME of an LSEG or of a group of LSEGs.

SEGMENT NAME - LSEGs are assigned segment names at translation time. These names serve two purposes:

1. They play a role at LINK time in determining which LSEGs are combined with other LSEGs.

2. They are used in assembly source code to specify groups.

CLASS NAME - LSEGs may optionally be assigned Class Names at translation time. Classes define a partition on LSEGs: two LSEGs are in the same class if they have the same Class Name.

The Microsoft Linker applies the following semantics to class names. The class name "CODE" or any class name whose suffix is "CODE" implies that all segments of said class contain only code and may be considered read-only. Such segments may be overlayed if the user specifies the module containing the segment as part of an overlay.

OVERLAY NAME - LSEGs may optionally be assigned an overlay name. The overlay name of an LSEG is ignored by MS-LINK (version 2.40 and later versions), but it is used by Intel Relocation and Linkage products.

COMPLETE NAME - The complete name of an LSEG consists of the Segment Name, Class Name, and Overlay Name. LSEGs from different modules will be combined if their Complete Names are identical.

6.3 MODULE IDENTIFICATION AND ATTRIBUTES

A module header record is always the first record in a module. It provides a module name.

In addition to a name, a module may have the attribute of being a main program as well as having a specified starting address. When linking multiple modules together, only one module with the main attribute should be given.

In summary, modules may or may not be main and may or may not have a starting address.

6.4 SEGMENT DEFINITION

A module is a collection of object code defined by a sequence of records produced by a translator. The object code represents contiguous regions of memory whose contents are determined at translation time. These regions are called LOGICAL SEGMENTS (LSEGs). A module defines the attributes of each LSEG. The SEGMENT DEFINITION RECORD (SEGDEF) is the vehicle by which all LSEG information (name, length, memory alignment, etc.) is maintained. The LSEG information is required when multiple LSEGs are combined and when segment addressability (See Section 6.5, "Segment Addressing") is established. The SEGDEF records are required to follow the first header record.

6.5 SEGMENT ADDRESSING

The 8086 addressing mechanism provides segment base registers from which a 64K-byte region of memory, called a FRAME, may be addressed. There is one code segment base register (CS), two data segment base registers (DS, ES), and one stack segment base register (SS).

The possible number of LSEGs that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would be the case in a modular program with many small data and/or code LSEGs.

Since such frequent loading of base registers is undesirable, it is a good strategy to collect many small LSEGs together into a single unit that will fit in one memory frame so that all the LSEGs may be addressed using the same base register value. This addressable unit is a GROUP and has been defined earlier in Section 6.2, "Definition of Terms."

To allow addressability of objects within a GROUP to be established, each GROUP must be explicitly defined in the module. The GROUP DEFINITION RECORD (GRPDEF) provides a list of constituent segments either by segment name or by segment attribute such as "the segment defining symbol FOO" or "the segments with class name ROM."

The GRPDEF records within a module must follow all SEGDEF records as GRPDEF records may reference SEGDEF records in defining a GROUP. The GRPDEF records must also precede all other records except header records, as the Linker must process them first.

6.6 SYMBOL DEFINITION

MS-LINK supports three different types of records that fall into the class of symbol definition records. The two most important types are PUBLIC NAMES DEFINITION RECORDs (PUBDEFs) and EXTERNAL NAMES DEFINITION RECORDS (EXTDEFs). These types are used to define globally visible procedures and data items and to resolve external references. In addition, TYPDEF records are used by MS-LINK for the allocation of communal variables (see Section 6.14 "Microsoft Type Representations for Communal Variables").

6.7 INDICES

"Index" fields occur throughout this document. An index is an integer that selects some particular item from a collection of such items. (List of examples: NAME INDEX, SEGMENT INDEX, GROUP INDEX, EXTERNAL INDEX, TYPE INDEX.)

Note

An index is normally a positive number. The index value zero is reserved, and may carry a special meaning dependent upon the type of index (e.g., a Segment Index of zero specifies the "Unnamed," absolute pseudo-segment; a Type Index of zero specifies the "Untyped type", which is different from "Decline to state").

In general, indices must assume values quite large (that is, much larger than 255). Nevertheless, a great number of object files will contain no indices with values greater than 50 or 100. Therefore, indices will be encoded in one or two bytes, as required.

The high-order (left-most) bit of the first (and possibly the only) byte determines whether the index occupies one byte or two. If the bit is 0, then the index is a number between 0 and 127, occupying one byte. If the bit is 1, then the index is a number between 0 and 32K-1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.

6.8 CONCEPTUAL FRAMEWORK FOR FIXUPS

A "fixup" is some modification to object code, requested by a translator, performed by the Linker, achieving address binding.

> **Note**
>
> This definition of "fixup" accurately represents the viewpoint maintained by the Linker. Nevertheless, the Linker can be used to achieve modifications of object code (i.e., "fixups") that do not conform to this definition. For example, the binding of code to either hardware floating point or software floating point subroutines is a modification to an operation code, where the operation code is treated as if it were an address. The previous definition of "fixup" is not intended to disallow or disparage object code modifications.

8086 translators specify a fixup by giving four data:

1. The place and type of a LOCATION to be fixed up.

2. One of two possible fixup MODEs.

3. A TARGET, which is a memory address to which LOCATION must refer.

4. A FRAME defining a context within which the reference takes place.

LOCATION - There are 5 types of LOCATION: a POINTER, a BASE, an OFFSET, a HIBYTE, and a LOBYTE.

The vertical alignment of the following figure illustrates four points. (Remember that the high-order byte of a word in 8086 memory is the byte with the higher address.)

1. A BASE is the high-order word of a pointer (and the Linker doesn't care if the low-order word of the pointer is present or not).

2. An OFFSET is the low-order word of a pointer (and the Linker doesn't care if the high-order word follows or not).

3. A HIBYTE is the high-order half of an OFFSET (and the Linker doesn't care if the low-order half precedes or not).

4. A LOBYTE is the low-order half of an OFFSET (and the Linker doesn't care if the high-order half follows or not).

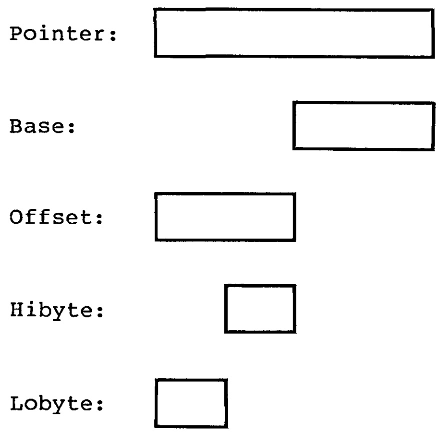


Figure 6.1 LOCATION Types

A LOCATION is specified by two data: (1) the LOCATION type, and (2) where the LOCATION is. The first is specified by the LOC subfield of the LOCAT field of the FIXUP record; the second is specified by the DATA RECORD OFFSET subfield of the LOCAT field of the FIXUP record.

MODE - The Linker supports two kinds of fixups: "self-relative" and "segment-relative."

Self-relative fixups support the 8- and 16-bit offsets that are used in the CALL, JUMP and SHORT-JUMP instructions. Segment-relative fixups support all other addressing modes of the 8086.

TARGET - The TARGET is the location in MAS being referenced. (More explicitly, the TARGET may be considered to be the lowest byte in the object being referenced.) A TARGET is specified in one of eight ways. There are four "primary" ways, and four "secondary" ways. Each primary way of specifying a TARGET uses two kinds of data: an INDEX-or-FRAME-NUMBER 'X', and a displacement 'D'.

(T0) X is a SEGMENT INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.

(T1) X is a GROUP INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.

(T2) X is an EXTERNAL INDEX. The TARGET is the Dth byte following the byte whose address is (eventually) given by the External Name identified by the INDEX.

(T3) X is a FRAME NUMBER. The TARGET is the Dth byte in the FRAME identified by the FRAME NUMBER (i.e., the address of TARGET is (X*16)+D).

Each secondary way of specifying a TARGET uses only one data item: the INDEX-or-FRAME-NUMBER X. An implicit displacement equal to zero is assumed.

(T4) X is a SEGMENT INDEX. The TARGET is the 0th (first) byte in the LSEG identified by the INDEX.

(T5) X is a GROUP INDEX. The TARGET is the 0th (first) byte in the LSEG in the specified group that is eventually LOCATEd lowest in MAS.

(T6) X is an EXTERNAL INDEX. The TARGET is the byte whose address is the External Name identified by the INDEX.

(T7) X is a FRAME NUMBER. The TARGET is the byte whose 20-bit address is (X*16).

Note

The Microsoft Linker does not support methods T3 and T7.

The following nomenclature is used to describe a TARGET:

TARGET: SI(<segment name>), <displacement> [T0]

TARGET: GI(<group name>), <displacement> [T1]

TARGET: EI(<symbol name>), <displacement> [T2]

TARGET: SI (<segment name>) [T4]

TARGET: GI (<group name>) [T5]

TARGET: EI (<symbol name>) [T6]

The following examples illustrate how this notation is used:

TARGET: SI(CODE), 1024	The 1025th byte in the segment "CODE".
TARGET: GI(DATAAREA)	The location in MAS of a group called "DATAAREA".
TARGET: EI(SIN)	The address of the external subroutine "SIN".
TARGET: EI(PAYSCHEDULE), 24	The 24th byte following the location of an EXTERNAL data structure called "PAYSCHEDULE".

FRAME - Every 8086 memory reference is to a location contained within some FRAME; where the FRAME is designated by the content of some segment register. For the Linker to form a correct, usable memory reference, it must know what the TARGET is, and to which FRAME the reference is being made. Thus, every fixup specifies such a FRAME, in one of six ways. Some ways use data, X, which is in INDEX-or-FRAME-NUMBER, as above. Other ways require no data.

The six ways of specifying frames are:

(F0) X is a SEGMENT INDEX. The FRAME is the canonic FRAME of the LSEG defined by the INDEX.

(F1) X is a GROUP INDEX. The FRAME is the canonic FRAME defined by the group (i.e., the canonic FRAME defined by the LSEG in the group that is eventually LOCATEd lowest in MAS).

(F2) X is an EXTERNAL INDEX. The FRAME is determined when the External Name's public definition is found. There are three cases:

- (F2a) The symbol is defined relative to some LSEG, and there is no associated GROUP. The LSEGs canonic FRAME is specified.

- (F2b) The symbol is defined absolutely, without reference to an LSEG, and there is no associated GROUP. The FRAME is specified by the FRAME NUMBER subfield of the PUBDEF record that gives the symbol's definition.

- (F2c) Regardless of how the symbol is defined, there is an associated GROUP. The canonic FRAME of the GROUP is specified. (The group is specified by the GROUP INDEX subfield of the PUBDEF Record.)

(F3) X is a FRAME NUMBER (specifying the obvious FRAME).

(F4) No X. The FRAME is the canonic FRAME of the LSEG containing LOCATION.

(F5) No X. The FRAME is determined by the TARGET. There are four cases:

- (F5a) The TARGET specified a SEGMENT INDEX: in this case, the FRAME is determined as in (F0).

- (F5b) The TARGET specified a GROUP INDEX: in this case, the FRAME is determined as in (F1).

- (F5c) The TARGET specified an EXTERNAL INDEX: in this case, the FRAME is determined as in (F2).

- (F5d) The TARGET is specified with an explicit FRAME NUMBER: in this case the FRAME is determined as in (F3).

Note

The Microsoft Linker does not support frame methods F2b, F3, and F5d.

Nomenclature describing FRAMEs is similar to the above nomenclature for TARGETs.

FRAME:	SI (<segment name>)	[F0]
FRAME:	GI (<group name>)	[F1]
FRAME:	EI (<symbol name>)	[F2]
FRAME:	LOCATION	[F4]
FRAME:	TARGET	[F5]
FRAME:	NONE	[F6]

For an 8086 memory reference, the FRAME specified by a self-relative reference is usually the canonic FRAME of the LSEG containing the LOCATION, and the FRAME specified by a segment relative reference is the canonic FRAME of the LSEG containing the TARGET.

6.9 SELF-RELATIVE FIXUPS

A self-relative fixup operates as follows: A memory address is implicitly defined by LOCATION; namely the address of the byte following LOCATION (because at the time of a self-relative reference, the 8086 IP (Instruction Pointer) is pointing to the byte following the reference).

For 8086 self-relative references, if either LOCATION or TARGET are outside the specified FRAME, the Linker gives a warning. Otherwise, there is a unique 16-bit displacement which, when added to the address implicitly defined by LOCATION, will yield the relative position of TARGET in the FRAME.

If the LOCATION is an OFFSET, the displacement is added to LOCATION modulo 65536; no errors are reported.

If the LOCATION is a LOBYTE, the displacement must be within the range {-128:127}, otherwise the Linker will give a warning. The displacement is added to LOCATION modulo 256.

If the LOCATION is a BASE, POINTER, or HIBYTE, it is unclear what the translator had in mind, and the action taken by the Linker is undefined.

6.10 SEGMENT-RELATIVE FIXUPS

A segment-relative fixup operates in the following way: a non-negative 16-bit number, FBVAL, is defined as the FRAME NUMBER of the FRAME specified by the fixup, and a signed 20-bit number, FOVAL, is defined as the distance from the base of the FRAME to the TARGET. If this signed 20-bit number is less than 0 or greater than 65535, the Linker reports an error. Otherwise, FBVAL and FOVAL are used to fixup LOCATION in the following fashion:

1. If LOCATION is a POINTER, then FBVAL is added (modulo 65536) to the high-order word of POINTER, and FOVAL is added (modulo 65536) to the low-order word of POINTER.

2. If LOCATION is a BASE, then FBVAL is added (modulo 65536) to the BASE; FOVAL is ignored.

3. If LOCATION is an OFFSET, then FOVAL is added (modulo 65536) to the OFFSET; FBVAL is ignored.

4. If LOCATION is a HIBYTE, then (FOVAL/256) is added (modulo 256) to the HIBYTE; FBVAL is ignored. (The indicated division is "integer division", i.e., the remainder is discarded.)

5. If LOCATION is a LOBYTE, then (FOVAL modulo 256) is added (modulo 256) to the LOBYTE; FBVAL is ignored.

6.11 RECORD ORDER

A object code file must contain a sequence of (one or more) modules, or a library containing zero or more modules. A module is defined as a collection of object code defined by a sequence of object records. The following syntax shows the valid orderings of records to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence.

> **Note**
>
> The syntactic description language used below is defined in WIRTH: CACM, November 1977, vol.#20, no.#11, pp.#822-823. The character strings represented by capital letters above are not literals but are identifiers that are further defined in the section describing the record formats.

```
object file   = tmodule

tmodule       = THEADR seg-grp   {component}   modtail

seg_grp       = {LNAMES} {SEGDEF} {TYPDEF | EXTDEF | GRPDEF}

component     = data | debug_record

data          = content_def | thread_def |
                     TYPDEF | PUBDEF | EXTDEF

debug_record  = LINNUM

content_def   = data_record {FIXUPP}

thread_def    = FIXUPP (containing only thread fields)

data_record   = LIDATA | LEDATA

modtail       = MODEND
```

The following rules apply:

1. A FIXUPP record always refers to the previous DATA record.

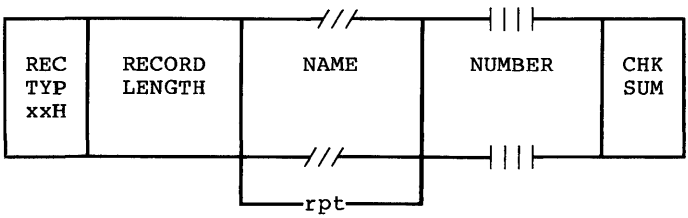
2. All LNAMES, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.

3. COMENT records may appear anywhere in a file, except as the first or last record in a file or module, or within a contentdef.

6.12 INTRODUCTION TO THE RECORD FORMATS

The following pages present diagrams of record formats in schematic form. Here is a sample record format, to illustrate the various conventions.

SAMPLE RECORD FORMAT
(SAMREC)



TITLE and OFFICIAL ABBREVIATION

At the top is the name of the record format described, with an official abbreviation. To promote uniformity among various programs, including translators and debuggers, the abbreviation should be used in both code and documentation. The record format abbreviation is always six letters.

The BOXES

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes represent two bytes each. The wide boxes with three slashes in the top and bottom represent a variable number of bytes, one or more, depending upon content. The wide boxes with four vertical bars in the top and bottom represent 4-byte fields.

RECTYP

The first byte in each record contains a value between 0 and 255, indicating which record type the record is.

RECORD LENGTH

The second field in each record contains the number of bytes in the record, exclusive of the first two fields.

NAME

Any field that indicates a "NAME" has the following internal structure: the first byte contains a number between 0 and 127, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string.

Most translators constrain the character set to be a subset of the ASCII character set.

NUMBER

A 4-byte NUMBER field represents a 32-bit unsigned integer, where the first 8 bits (least-significant) are stored in the first byte (lowest address), the next 8 bits are stored in the second byte, and so on.

REPEATED OR CONDITIONAL FIELDS

Some portions of a record format contain a field or a series of fields that may be repeated one or more times. Such portions are indicated by the "repeated" or "rpt" brackets below the boxes.

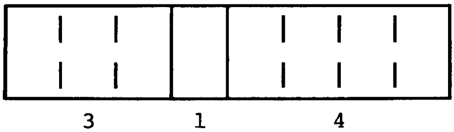
Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar "conditional" or "cond" brackets below the boxes.

CHKSUM

The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

BIT FIELDS

Descriptions of contents of fields will sometimes be at the bit level. Boxes with vertical lines drawn through them represent bytes or words; the vertical lines indicate bit boundaries; thus the byte represented below, has three bit-fields of 3-, 1-, and 4-bits.



T-MODULE HEADER RECORD
(THEADR)

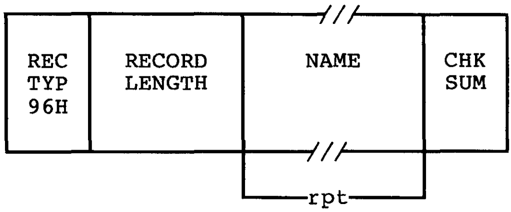
REC TYP 80H	RECORD LENGTH	T MODULE NAME	CHK SUM

Every module output from a translator must have a T-MODULE HEADER RECORD.

T-MODULE NAME

The T-MODULE NAME provides a name for the T-MODULE.

LIST OF NAMES RECORD
(LNAMES)



This Record provides a list of names that may be used in following SEGDEF and GRPDEF records as the names of Segments, Classes and/or Groups.

The ordering of LNAMES records within a module, together with the ordering of names within each LNAMES Record, induces an ordering on the names. Thus, these names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as "Name Indices" in the Segment Name Index, Class Name Index and Group Name Index fields of the SEGDEF and GRPDEF Records.

NAME

This repeatable field provides a name, which may have zero length.

SEGMENT DEFINITION RECORD
(SEGDEF)

REC TYP 98H	RECORD LENGTH	SEGMENT ATTR	SEGMENT LENGTH	SEGMENT NAME INDEX	CLASS NAME INDEX	OVER LAY NAME INDEX	CHK SUM

SEGMENT INDEX values 1 through 32767, which are used in other record types to refer to specific LSEGs, are defined implicitly by the sequence in which SEGDEF Records appear in the object file.

SEG ATTR

The SEG ATTR field provides information on various attributes of a segment, and has the following format:

ACB P	FRAME NUMBER	OFF SET
	conditional	

The ACBP byte contains four numbers which are the A, C, B, and P attribute specifications. This byte has the following format:

A	C	B	P

"A" (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

A=0 SEGDEF describes an absolute LSEG.
A=1 SEGDEF describes a relocatable, byte-aligned LSEG.

A=2 SEGDEF describes a relocatable, word-aligned LSEG.
A=3 SEGDEF describes a relocatable, paragraph-aligned LSEG.
A=4 SEGGDEF describes a relocatable, page-aligned LSEG.

If A=0, the FRAME NUMBER and OFFSET fields will be present. Using MS-LINK, absolute segments may be used for addressing purposes only; for example, defining the starting address of a ROM and defining symbolic names for addresses within the ROM. MS-LINK will ignore any data specified as belonging to an absolute LSEG.

"C" (Combination) is a 3-bit subfield that specifies the combination attribute of the LSEG. Absolute segments (A=0) must have combination zero (C=0). For relocatable segments, the C field encodes a number (0,1,2,4,5,6 or 7) that indicates how the segment can be combined. The interpretation of this attribute is best given by considering how two LSEGs are combined: Let X,Y be LSEGs, and let Z be the LSEG resulting from the combination of X,Y. Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX, LY. Let G be the length of any gap required between the X- and Y-components of Z to accommodate the alignment attribute of Y. Let LZ denote the length of the (combined) LSEG Z; let dx (0<=dx<LX) be the offset in X of a byte, and let dy similarly be the offset in Y of a byte. The following table gives the length LZ of the combined LSEG Z, and the offsets dx' and dy' in Z for the bytes corresponding to dx in X and dy in Y. Intel defines additionally alignment types 5 and 6 and also processes code and data placed in segment with align-type.

Table 6.2 Combination Attribute Example

C	LZ	dx'	dy'	
2	LX+LY+G	dx	dy+LX+G	"Public"
5	LX+LY+G	dx	dy+LX+G	"Stack"
6	MXY	dx	dy	"Common"

Table 6.2 has no lines for C=0, C=1, C=3, C=4 and C=7. C=0 indicates that the relocatable LSEG may not be combined; C=1 and C=3 are undefined. C=4 and C=7 are treated like C=2. C1, C4, and C7 all have different meanings according to the Intel standard.

"B" (Big) is a 1-bit subfield which, if 1, indicates that the Segment Length is exactly 64K (65536). In this case the SEGMENT LENGTH field must contain zero.

The "P" field must always be zero. The "P" field is the "Page resident" field in Intel-Land.

The FRAME NUMBER and OFFSET fields (present only for absolute segments, A=0) specify the placement in MAS of the absolute segment. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET, then an adjustment of the FRAME NUMBER should be done.

SEGMENT LENGTH

The SEGMENT LENGTH field gives the length of the segment in bytes. The length may be zero; if so, MS-LINK will not delete the segment from the module. The SEGMENT LENGTH field is only big enough to hold numbers from 0 to 64K-1 inclusive. The B attribute bit in the ACBP field (see SEG ATTR section) must be used to give the segment a length of 64K.

SEGMENT NAME INDEX

The Segment Name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, TAXDATA, MODULENAME_CODE, STACK. This field provides the Segment Name, by indexing into the list of names provided by the LNAMES Record(s).

CLASS NAME INDEX

The Class Name is a name the programmer or translator can assign to a segment. If none is assigned, the name is null, and has length 0. The purpose of Class Names is to allow the programmer to define a "handle" used in the ordering of the LSEGs in MAS. Examples: RED, WHITE, BLUE; ROM FASTRAM, DISPLAYRAM. This field provides the Class Name, by indexing into the list of names provided by the LNAMES Record(s).

OVERLAY NAME INDEX

> **Note**
>
> This is ignored in MS-LINK versions 2.40 and later, but supported in all earlier versions. However, semantics differ from Intel semantics.

The Overlay Name is a name the translator and/or MS-LINK, at the programmer's request, applies to a segment. The Overlay Name, like the Class Name, may be null. This field provides the Overlay Name, by indexing into the list of names provided by the LNAMES Record(s).

> **Note**
>
> The "Complete Name" of a segment is a 3-component entity comprising a Segment Name, a Class Name and an Overlay Name. (The latter two components may be null.)

GROUP DEFINITION RECORD
(GRPDEF)

REC TYP 9AH	RECORD LENGTH	GROUP NAME INDEX	GROUP COMPONENT DESCRIPTOR	CHK SUM
			repeated	

GROUP NAME INDEX

The Group Name is a name by which a collection of LSEGs may be referenced. The important property of such a group is that, when the LSEGs are eventually fixed in MAS, there must exist some FRAME which "covers" every LSEG of the group.

The GROUP NAME INDEX field provides the Group Name, by indexing into the list of names provided by the LNAMES Record(s).

GROUP COMPONENT DESCRIPTOR

Each GROUP COMPONENT DESCRIPTOR has the following format:

SI (FFH)	SEGMENT INDEX

The first byte of the DESCRIPTOR contains 0FFH; the DESCRIPTOR contains one field, which is a SEGMENT INDEX that selects the LSEG described by a preceding SEGDEF record.

Intel defines 4 other group descriptor types, each with its own meaning. They are 0FEH, 0FDH, 0FBH, and 0FAH. The Microsoft Linker will treat all of these values the same as 0FFH (i.e., it always expects 0FFH followed by a segment index, and it does not, in fact, check to see if the value is actually 0FF).

TYPE DEFINITION RECORD
(TYPDEF)

REC TYP 8EH	RECORD LENGTH	NAME (USUALLY NULL)	EIGHT LEAF DESCRIPTOR	CHK SUM
			repeated	

The Microsoft Linker uses TYPDEF records only for communal variable allocation. This is not Intel's intended purpose. See Section 6.14, "Microsoft Type Representations for Communal Variables."

As many "EIGHT LEAF DESCRIPTOR" fields as necessary are used to describe a branch. (Every such field except the last in the record describes eight leaves; the last such field describes from one to eight leaves.)

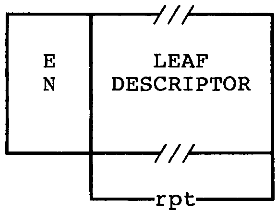
TYPE INDEX values 1 through 32767, which are contained in other record types to associate object types with object names, are defined implicitly by the sequence in which TYPDEF records appear in the object file.

NAME

Use of this field is reserved. Translators should place a single byte containing 0 in it (which is the representation of a name of length zero).

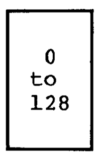
EIGHT LEAF DESCRIPTOR

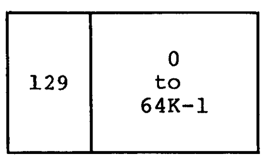
This field can describe up to eight Leaves.



The EN field is a byte: the 8 bits, left to right, indicate if the following 8 Leaves (left to right) are Easy (bit=0) or Nice (bit=1).

The LEAF DESCRIPTOR field, which occurs between 1 and 8 times, has one of the following formats:





132	0 to 16M-1

136	-2G-1 to 2G-1

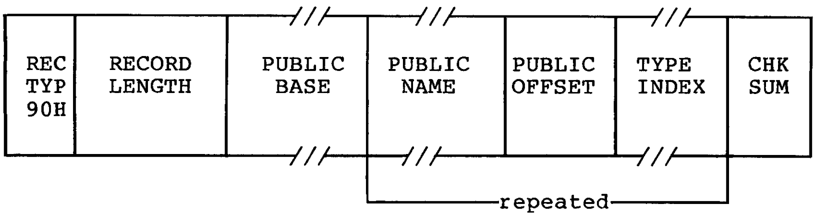
The first format (single byte), containing a value between 0 and 127, represents a Numeric Leaf whose value is the number given.

The second format, with a leading byte containing 129, represents a Numeric Leaf. The number is contained in the following two bytes.

The third format, with a leading byte containing 132, represents a Numeric Leaf. The number is contained in the following three bytes.

The fourth format, with a leading byte containing 136, represents a Signed Numeric Leaf. The number is contained in the following four bytes, sign extended if necessary.

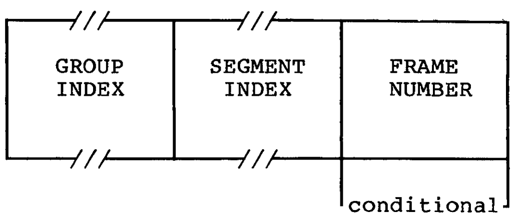
PUBLIC NAMES DEFINITION RECORD
(PUBDEF)



This record provides a list of one or more PUBLIC NAMEs; for each one, three data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

PUBLIC BASE

The PUBLIC BASE has the following format:



The GROUP INDEX field has a format given earlier, and provides a number between 0 and 32767 inclusive. A non-zero GROUP INDEX associates a group with the public symbol, and is used as described in Section 6.8, "Conceptual Framework for Fixups," case (F2c). A zero GROUP INDEX indicates that there is no associated group.

The SEGMENT INDEX field has a format given earlier, and provides a number between 0 and 32767, inclusive.

A non-zero SEGMENT INDEX selects an LSEG. In this case, the location of each public symbol defined in the record is taken as a non-negative displacement (given by a PUBLIC OFFSET field) from the first byte of the selected LSEG, and the FRAME NUMBER field must be absent.

A SEGMENT INDEX of 0 (legal only if GROUP INDEX is also 0) means that the location of each public symbol defined in the record is taken as a displacement from the base of the FRAME defined by the value in the FRAME NUMBER field.

The FRAME NUMBER is present if both the SEGMENT INDEX and GROUP INDEX are zero.

A non-zero GROUP INDEX selects some group; this group is taken as the "frame of reference" for references to all public symbols defined in this record; that is, MS-LINK will perform the following actions:

1. Any fixup of the form:

   TARGET: EI(P)

   FRAME: TARGET

   (where "P" is a public symbol in this PUBDEF record) will be converted by MS-LINK to a fixup of the form:

   TARGET: SI(L),d

   FRAME: GI(G)

   where "SI(L)" and "d" are provided by the SEGMENT INDEX and PUBLIC OFFSET fields. (The "normal" action would have the frame specifier in the new fixup be the same as in the old fixup: FRAME: TARGET.)

2. When the value of a public symbol, as defined by the SEGMENT INDEX, PUBLIC OFFSET, and (optionally) FRAME NUMBER fields, is converted to a {base,offset} pair, the base part will be taken as the base of the indicated group. If a non-negative 16-bit offset cannot then complete the definition of the public symbol's value, an error occurs.

A GROUP INDEX of zero selects no group. MS-LINK will not alter the FRAME specification of fixups referencing the symbol, and will take, as the base part of the absolute value of the public symbol, the canonic frame of the segment (either LSEG or PSEG) determined by the SEGMENT INDEX field.

PUBLIC NAME

The PUBLIC NAME field gives the name of the object whose location in MAS is made available to other modules. The name must contain one or more characters.

PUBLIC OFFSET

The PUBLIC OFFSET field is a 16-bit value, which is either the offset of the Public Symbol with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the Public Symbol with respect to the specified FRAME (if SEGMENT INDEX = 0).

TYPE INDEX

The TYPE INDEX field identifies a single preceding TYPDEF (Type Definition) Record containing a descriptor for the type of entity represented by the Public Symbol. This field is ignored by the Linker.

EXTERNAL NAMES DEFINITION RECORD
(EXTDEF)

REC TYP 8CH	RECORD LENGTH	EXTERNAL NAME	TYPE INDEX	CHK SUM
		repeated	repeated	

This record provides a list of external names, and for each name, the type of object it represents. MS-LINK will assign to each External Name the value provided by an identical Public Name (if such a name is found).

EXTERNAL NAME

This field provides the name, which must have non-zero length, of an external object.

Inclusion of a Name in an External Names Record is an implicit request that the object file be linked to a module containing the same name declared as a Public Symbol. This request obtains whether or not the External Name is referenced within some FIXUPP Record in the module.

The ordering of EXTDEF Records within a module, together with the ordering of External Names within each EXTDEF Record, induces an ordering on the set of all External Names requested by the module. Thus, External Names are considered to be numbered 1, 2, 3, 4, ... These numbers are used as "External Indices" in the TARGET DATUM and/or FRAME DATUM fields of FIXUPP Records to refer to a particular External Name.

Note

8086 External Names are numbered positively: 1,2,3,... This is a change from 8080 External Names, which were numbered starting from zero: 0,1,2,... This conforms with other 8086 Indices (Segment Index, Type Index, etc.) which use 0 as a default value with special meaning.

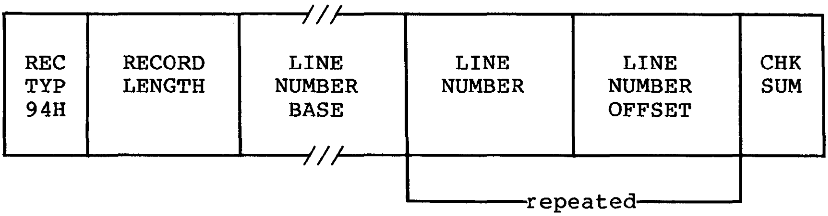
External indices may not reference forward. For example, an external definition record defining the kth object must precede any record referring to that object with index k.

TYPE INDEX

This field identifies a single preceding TYPDEF (Type Definition) record containing a descriptor for the type of object named by the External Symbol.

The TYPE INDEX is used only in communal variable allocation by the Microsoft Linker.

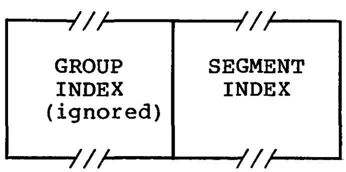
LINE NUMBERS RECORD
(LINNUM)



This record provides the means by which a translator may pass the correspondence between a line number in source code and the corresponding translated code.

LINE NUMBER BASE

The LINE NUMBER BASE has the following format:



The SEGMENT INDEX determines the location of the first byte of code corresponding to some source line number.

LINE NUMBER

A line number between 0 and 32767, inclusive, is provided in binary by this field. The high-order bit is reserved for future use and must be zero.

LINE NUMBER OFFSET

The LINE NUMBER OFFSET field is a 16-bit value, which is the offset of the line number with respect to an LSEG (if SEGMENT INDEX > 0).

LOGICAL ENUMERATED DATA RECORD
(LEDATA)

REC TYP A0H	RECORD LENGTH	SEGMENT INDEX	ENUMERATED DATA OFFSET	DAT	CHK SUM
				rpt	

This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

SEGMENT INDEX

This field must be non-zero and specifies an index relative to the SEGMENT DEFINITION RECORDS found previous to the LEDATA RECORD.

ENUMERATED DATA OFFSET

This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory.

DAT

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

LOGICAL ITERATED DATA RECORD
(LIDATA)

REC TYP A2H	RECORD LENGTH	SEGMENT INDEX	ITERATED DATA OFFSET	ITERATED DATA BLOCK	CHK SUM
				repeated	

This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

SEGMENT INDEX

This field must be non-zero and specifies an index relative to the SEGDEF records found previous to the LIDATA RECORD.

ITERATED DATA OFFSET

This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory.

ITERATED DATA BLOCK

This repeated field is a structure specifying the repeated data bytes. The structure has the following format:

REPEAT COUNT	BLOCK COUNT	CONTENT

Note

The Linker cannot handle LIDATA records whose ITERATED DATA BLOCK is larger than 512 bytes.

REPEAT COUNT

This field specifies the number of times that the CONTENT portion of this ITERATED DATA BLOCK is to be repeated. REPEAT COUNT must be non-zero.

BLOCK COUNT

This field specifies the number of ITERATED DATA BLOCKS that are to be found in the CONTENT portion of this ITERATED DATA BLOCK. If this field has value zero, then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as data bytes. If non-zero, then the CONTENT portion is interpreted as that number of ITERATED DATA BLOCKs.

CONTENT

This field may be interpreted in one of two ways, depending on the value of the previous BLOCK COUNT field.

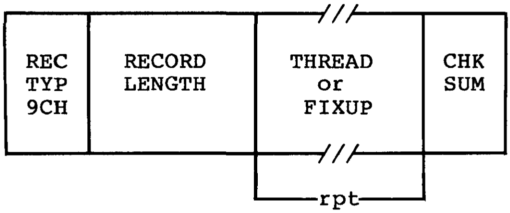
If BLOCK COUNT is zero, then this field is a 1-byte count followed by the indicated number of data bytes.

If BLOCK COUNT is non-zero, then this field is interpreted as the first byte of another ITERATED DATA BLOCK.

Note

From the outermost level, the number of nested ITERATED DATA BLOCKS is limited to 17, i.e., the number of levels of recursion is limited to 17.

FIXUP RECORD
(FIXUPP)

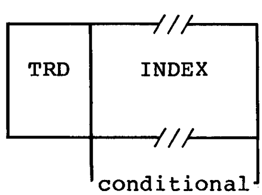


This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a LOCATION within the previous DATA record. A data record may be followed by more than one fixup record that refers. Each fixup is specified by a FIXUP field that specifies four data: a location, a mode, a target and a frame. The frame and the target may be specified totally within the FIXUP field, or may be specified by reference to a preceding THREAD field.

A THREAD field specifies a default target or frame that may subsequently be referred to in identifying a target or a frame. Eight threads are provided; four for frame specification and four for target specification. Once a target or frame has been specified by a THREAD, it may be referred to by following FIXUP fields (in the same or following FIXUPP records), until another THREAD field with the same type (TARGET or FRAME) and Thread Number (0 - 3) appears (in the same or another FIXUPP record).

THREAD

THREAD is a field with the following format:



The TRD DAT (ThReaD DATa) subfield is a byte with this internal structure:

0	D	Z	METHOD	THRED

The "Z" is a 1-bit subfield, currently without any defined function, that is required to contain 0.

The "D" subfield is one bit that identifies what type of thread is being specified. If D=0, then a target thread is being defined; if D=1, then a frame thread is being defined.

METHOD is a 3-bit subfield containing a number between 0 and 3 (D=0) or a number between 0 and 6 (D=1).

If D=0, then METHOD = (0, 1, 2, 3, 4, 5, 6, 7) mod 4, where the 0, ..., 7 indicate methods T0, ..., T7 of specifying a target. Thus, METHOD indicates what kind of Index or Frame Number is required to specify the target, without indicating if the target will be specified in a primary or secondary way. Note that methods 2b, 3, and 7 are not supported by MS-LINK.

If D=1, then METHOD = 0, 1, 2, 4, 5, corresponding to methods F0, ..., of specifying a frame. Here, METHOD indicates what kind (if any) of Index is required to specify the frame. Note that methods 3 and 5d are not supported by MS-LINK.

THRED is a number between 0 and 3, and associates a Thread Number to the frame or target defined by the THREAD field.

INDEX contains a Segment Index, Group Index, or External Index depending on the specification in the METHOD subfield. This subfield will not be present if F4 or F5 are specified by METHOD.

FIXUP

FIXUP is a field with the following format:

LOCAT	FIX DAT	FRAME DATUM	TARGET DATUM	TARGET DIS-PLACEMENT
		conditional	conditional	conditional

LOCAT is a byte pair with the following format:

1	M	S	LOC	DATA RECORD OFFSET
lo byte				hi byte

M is a 1-bit subfield that specifies the mode of the fixups: self-relative (M=0) or segment-relative (M=1).

> **Note**
>
> Self-relative fixups may not be applied to LIDATA records.

"S" is a 1-bit subfield that specifies that the length of the TARGET DISPLACEMENT subfield. If it is present in this FIXUP field (see below), it will be either two bytes (containing a 16-bit non-negative number, S=0) or three bytes (containing a signed 24-bit number in 2's complement form, S=1).

> **Note**
>
> 3-byte subfields are a possible future extension, and are not currently supported. Thus, S=0 is currently mandatory.

LOC is a 3-bit subfield indicating that the byte(s) in the preceding DATA Record to be fixed up are a "lobyte" (LOC=0), an "offset" (LOC=1), a "base" (LOC=2), a "pointer" (LOC=3), or a "hibyte" (LOC=4). Other values in LOC are invalid.

The DATA RECORD OFFSET is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of LOCATION (the actual bytes being fixed up) within the preceding DATA record. The DATA RECORD OFFSET is relative to the first byte in the data fields in the DATA RECORDs.

> **Note**
>
> If the preceding DATA record is an LIDATA record, it is possible for the value of DATA RECORD OFFSET to designate a "location" within a REPEAT COUNT subfield or a BLOCK COUNT subfield of the ITERATED DATA field. Such a reference is an error. MS-LINK's action on such a malformed record is undefined.

FIX DAT is a byte with the following format:

F	FRAME	T	P	TARGT
	See Note 1			See Note 2

Note 1: Frame method 2b, F3, and F5d are not supported.

Note 2: Target method T3 and T7 are not supported.

F is a 1-bit subfield that specifies whether the frame for this FIXUP is specified by a thread (F=1) or explicitly (F=0).

FRAME is a number interpreted in one of two ways as indicated by the F bit. If F is zero, FRAME is a number between 0 and 5 and corresponds to methods F0, ..., F5 of specifying a FRAME. If F=1, then FRAME is a thread number (0-3). It specifies the frame most recently defined by a THREAD field that defined a frame thread with the same thread number. (Note that the THREAD field may appear in the same, or in an earlier FIXUPP record.)

"T" is a 1-bit subfield that specifies whether the target specified for this fixup is defined by reference to a thread (T=1), or is given explicitly in the FIXUP field (T=0).

"P" is a 1-bit subfield that indicates whether the target is specified in a primary way (requires a TARGET DISPLACEMENT, P=0) or specified in a secondary way (requires no TARGET DISPLACEMENT, P=1). Since a target thread does not have a primary/secondary attribute, the P bit is the only field that specifies the primary/secondary attribute of the target specification.

TARGT is interpreted as a 2-bit subfield. When T=0, it provides a number between 0 and 3, corresponding to methods T0, ..., T3 or T4, ..., T7, depending on the value of P (P can be interpreted as the high-order bit of T0, ..., T7). When the target is specified by a thread (T=1), then TARGT specifies a thread number (0-3).

FRAME DATUM is the "referent" portion of a frame specification, and is a Segment Index, a Group Index, an External Index. The FRAME DATUM subfield is present only when the frame is specified neither by a thread (F=0) nor explicitly by methods F4 or F5 or F6.

TARGET DATUM is the "referent" portion of a target specification, and is a Segment Index, a Group Index, an External Index or a Frame Number. The TARGET DATUM subfield is present only when the target is not specified by a thread (T=0).

TARGET DISPLACEMENT is the 2-byte displacement required by "primary" ways of specifying TARGETs. This 2-byte subfield is present if P=0.

> **Note**
>
> All these methods are described in Section 6.8, "Conceptual Framework for Fixups."

MODULE END RECORD
(MODEND)

REC TYP 8AH	RECORD LENGTH	MOD TYP	START ADDRS	CHK SUM

+conditional+

This record serves two purposes. It denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true, the execution address is specified.

MOD TYP

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:

MATTR	Z	Z	Z	Z	Z	L

MATTR is a 2-bit subfield that specifies the following module attributes:

MATTR	**MODULE ATTRIBUTE**
0	Non-main module with no START ADDRS
1	Non-main module with START ADDRS
2	Main module with no START ADDRS
3	Main module with START ADDRS

"L" indicates whether the START ADDRS field is interpreted as a logical address that requires fixing up by MS-LINK. (L=1). Note: with MS-LINK, L must always equal 1.

"Z" indicates that this bit has not currently been assigned a function. These bits are required to be zero.

Physical start addresses (L=0) are not supported.

The START ADDRS field (present only if MATTR is 1 or 3) has the following format:

START ADDRS

END DAT	FRAME DATUM	TARGET DATUM	TARGET DIS-PLACEMENT
	+conditional+	conditional+	conditional+

The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of a logical starting address to a physical starting address is done in exactly the same manner as mapping any other logical address to a physical address as specified in the discussion of fixups and the FIXUPP record. The above subfields of the START ADDRS field have the same semantics as the FIX DAT, FRAME DATUM, TARGET DATUM, and TARGET DISPLACEMENT fields in the FIXUPP record. Only "primary" fixups are allowed. Frame method F4 is not allowed.

COMMENT RECORD
(COMENT)

REC TYP 88H	RECORD LENGTH	COMMENT TYPE	COMMENT	CHK SUM

This record allows translators to include comments in object text.

COMMENT TYPE

This field indicates the type of comment carried by this record. This allows comments to be structured for those processes that wish to selectively act on comments. The format of this field is as follows:

N P	N L	Z	Z	Z	Z	Z	Z	COMMENT CLASS

The NP (NOPURGE) bit, if 1, indicates that it is not able to be purged by object file utility programs which implement the capability of deleting COMENT record.

The NL (NOLIST) bit, if 1, indicates that the text in the COMMENT field is not to be listed in the listing file of object file utility programs which implement the capability of listing object COMENT records.

The COMMENT CLASS field is defined as follows:

0	Language translator comment.
1	Intel copyright comment. The NP bit must be set.
2-155	Reserved for Intel use. (See note 1 below.)
156-255	Reserved for users. Intel products will apply no semantics to these values. (See Note 2 below.)

COMMENT

This field provides the commentary information.

Notes:

1. Class value 129 is used to specify a library to add to the Linker's library search list. The comment field will contain the name of the library. Note that unlike all other name specifications, the library name is not prefixed with its length. Its length is determined by the record length. The "NODEFAULTLIBRARYSEARCH" switch causes the linker to ignore all comment records whose class value is 129.

2. Class value 156 is used to specify a DOS level number. When the class value is 156, the comment field will contain a two-byte integer specifying a DOS level number.

6.13 NUMERIC LIST OF RECORD TYPES

```
*6E RHEADR
*70 REGINT
*72 REDATA
*74 RIDATA
*76 OVLDEF
*78 ENDREC
*7A BLKDEF
*7C BLKEND
*7E DEBSYM
 80 THEADR
*82 LHEADR
*84 PEDATA
*86 PIDATA
 88 COMENT
 8A MODEND
 8C EXTDEF
 8E TYPDEF
 90 PUBDEF
*92 LOCSYM
 94 LINNUM
 96 LNAMES
 98 SEGDEF
 9A GRPDEF
 9C FIXUPP
*9E (none)
 A0 LEDATA
 A2 LIDATA
*A4 LIBHED
*A6 LIBNAM
*A8 LIBLOC
*AA LIBDIC
```

Note

Record types preceded by an asterisk (*) are not supported by the Microsoft Linker. They will be ignored if they are found in an object module.

6.14 MICROSOFT TYPE REPRESENTATIONS FOR COMMUNAL VARIABLES

This section defines the Microsoft standard for communal variable allocation on the 8086 and 80286.

A communal variable is an uninitialized public variable whose final size and location are not fixed at compile time. Communal variables are similar to FORTRAN common blocks in that if a communal variable is declared in more than one object module being linked together, then its actual size will be the largest size specified in the several declarations. In the C language, all uninitialized public variables are communal. The following example shows three different declarations of the same C communal variable:

```
char    foo[4];         /* In file a.c */
char    foo[1];         /* In file b.c */
char    foo[1024];      /* In file c.c */
```

If the objects produced from a.c, b.c, and c.c are linked together, then the linker will allocate 1024 bytes for the char array "foo".

A communal variable is defined in the object text by an external definition record (EXTDEF) and the type definition record (TYPDEF) to which it refers.

The TYPDEF for a communal variable has the following format:

REC TYP 8EH	RECORD LENGTH	0	EIGHT LEAF DESCRIPTOR	CHK SUM

The EIGHT LEAF DESCRIPTOR field has the following format:

E N	LEAF DESCRIPTOR

The EN field specifies whether the next 8 leaves in the LEAF DESCRIPTOR field are EASY (bit = 0) or NICE (bit = 1). This byte is always zero for TYPDEFS for communal variables.

The LEAF DESCRIPTOR field has one of the following two formats. The format for communal variables in the default data segment (near variables) is as follows:

NEAR 62H	VAR TYP	LENGTH IN BITS	VAR SUBTYP
			(optional)

The VARiable TYPe field may be either SCALAR (7BH), STRUCT (79H), or ARRAY (77H). The VAR SUBTYP field (if any) is ignored by the Linker. The format for communal variables not in the default data segment (far variables) is as follows:

FAR 61H	VAR TYP 77H	NUMBER OF ELEMENTS	ELEMENT TYPE INDEX

The VARiable TYPe field must be ARRAY (77H). The length field specifies the NUMBER OF ELEMENTS, and the ELEMENT TYPE INDEX is an index to a previously defined TYPDEF whose format is that of a near communal variable.

The format for the LENGTH IN BITS or NUMBER OF ELEMENTS fields is the same as the format for the LEAF DESCRIPTOR field, described in the TYPDEF record format section of this manual.

Link time semantics:

All EXTDEFs referencing a TYPDEF of one of the previously described formats are treated as communal variables. All others are treated as externally defined symbols for which a matching public symbol definition (PUBDEF) is expected. A PUBDEF matching a communal variable definition will override the communal variable definition. Two communal variable definitions are said to match if the names given in the definitions match. If two matching definitions disagree about whether a communal variable is near or far, the linker will assume the variable is near.

If the variable is near, then its size is the largest of the sizes specified for it. If the variable is far, then the Linker issues a warning if there are conflicting array element size specifications; if there are no such conflicts, then the variable's size is the element size times the largest number of elements specified. The sum of the sizes of all near variables must not exceed 64K bytes. The sum of the sizes of all far variables must not exceed the size of the machine's addressable memory space.

"Huge" communal variables:

A far communal variable whose size is larger than 64K bytes will reside in segments that are contiguous (8086) or have consecutive selectors (80286). No other data items will reside in the segments occupied by a huge communal variable.

If the linker finds matching huge and near communal variable definitions, it issues a warning message, since it is impossible for a near variable to be larger than 64K bytes.

Chapter 7
Programming Hints

CHAPTER 7

PROGRAMMING HINTS

7.1 INTRODUCTION

This chapter describes recommended MS-DOS 3.1 programming procedures. By using these programming hints, you can ensure compatibility with future versions of MS-DOS.

The hints are organized into the following categories:

Interrupts

System Calls

Device Management

Memory Management

Process Management

File and Directory Management

Miscellaneous

7.2 INTERRUPTS

Never explicitly issue Interrupt 22H (Terminate Process Exit Address).

> This should only be done by the DOS. To change the terminate address, use Function 35H (Get Interrupt Vector) to get the current address and save it, then use Function 25H (Set Interrupt Vector) to change the Interrupt 22H entry in the vector table to point to the new terminate address.

Use Interrupt 24H (Critical Error Handler Address) with care.

> The Interrupt 24H handler must preserve the ES register.
>
> Only system calls 01H-0CH can be made by an Interrupt 24H handler. Making any other calls will destroy the MS-DOS stack and prevent successful use of the Retry or Ignore options.
>
> The registers SS, SP, DS, BX, CX, and DX must be preserved when using the Retry or Ignore options.

When an Interrupt 24H (Critical Error Handler Address) is received, always IRET back to MS-DOS with one of the standard responses.

> Programs that do not IRET from Interrupt 24H leave the system in an unpredictable state until a function call other than 01H-0CH is made. The Ignore option may leave data in internal system buffers that is incorrect or invalid.

Avoid trapping Interrupt 23H (Control-C Handler Address) and Interrupt 24H (Critical Error Handler Address). Don't rely on trapping errors via Interrupt 24H as part of a copy protection scheme.

> These might not be included in future releases of the operating system.

Interrupt 23H (Control-C Handler Address) must never be issued by a user program.

> Interrupt 23H must be issued only by MS-DOS.

Save any registers your program uses before issuing Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write).

> These interrupts destroy all registers except for the segment registers.
>
> Avoid writing or reading an interrupt vector directly to or from memory.

Use Functions 25H and 35H (Set Interrupt Vector and Get Interrupt Vector) to set and get values in the interrupt table.

7.3 SYSTEM CALLS

Use new system calls.

Avoid using system calls that have been superseded by new calls unless a program must maintain backward compatibility with pre-2.0 versions of MS-DOS. See Section 1.8, "Old System Calls," of this manual for a list of these new calls.

Avoid using system calls 01H-0CH and 26H (Create New PSP).

Use the new "tools" approach for reading and writing on standard input and output. Use Function 4B00H (Load and Execute Program) instead of 26H to execute a child process.

Use file-sharing calls if more than one process is in effect.

See "File Sharing," in Section 1.5.2, "File-Related Function Requests" in Chapter 1 for more information.

Use networking calls where appropriate.

Some forms of IOCTL can only be used with Microsoft Networks. See Section 1.6, "Microsoft Networks," in this manual for a list of these calls.

When selecting a disk with Function 0EH (Select Disk), treat the value returned in AL with care.

The value in AL specifies the maximum number of logical drives; it does not specify which drives are valid.

7.4 DEVICE MANAGEMENT

Use installable device drivers.

MS-DOS provides a modular device driver structure for the BIOS, allowing you to configure and install device drivers at boot time. Block device drivers transmit a block of data at a time, while character device drivers transmit a byte of data at a time.

Examples of both types of device drivers are given in Chapter 2, "MS-DOS Device Drivers."

Use buffered I/O.

The device drivers can handle streams of data up to 64K. When sending a large amount of output to the screen, you can send it with one system call. This will increase performance.

Programs that use direct console I/O via Function 06H and 07H (Direct Console I/O and Direct Console Input) and that want to read Control-C as data should ensure that Control-C checking is off.

The program should ensure that Control-C checking is off by using Function 33H (Control-C Check).

Be compatible with international support.

To provide support for international character sets, MS-DOS recognizes all possible byte values as significant characters in filenames and data streams. Pre-2.x versions ignored the high bit in the MS-DOS filename.

7.5 MEMORY MANAGEMENT

Use memory management.

MS-DOS keeps track of allocated memory by writing a memory control block at the beginning of each area of memory. Programs should use Functions 48H (Allocate Memory), 49H (Free Allocated Memory), and 4AH (Set Block) to release unneeded memory.

This will allow for future compatibility.

See Section 1.3, "Memory Management," for more information.

Only use allocated memory.

Don't directly access memory that was not provided as a result of a system call. Do not use fixed addressing, use only relative references.

A program that uses memory that has not been allocated to it may destroy other memory control blocks or cause other applications to fail.

7.6 PROCESS MANAGEMENT

Use the EXEC Function Call to load and execute programs.

The EXEC Function (4B00H) is the preferred way to load programs and program overlays. Using the EXEC call instead of hard-coding information about how to load an .EXE file (or always assuming that your file is a .COM file) will isolate your program from changes in future releases of MS-DOS and .EXE file formats.

Use Function 31H (Keep Process), instead of Interrupt 27H (Terminate But Stay Resident). Function 31H allows programs to terminate and stay resident that are greater than 64K.

Programs should terminate using End Process (4CH).

Programs that terminate by

- a long jump to offset 0 in the PSP,
- issuing an Interrupt 20H with CS:0 pointing at the PSP,
- issuing an Interrupt 21H with AH=0, CS:0 pointing at the PSP, or
- a long call to location 50H in the PSP with AH=0

must ensure that the CS register contains the segment address of the PSP.

7.7 FILE AND DIRECTORY MANAGEMENT

Use the MS-DOS file management system.

Using the MS-DOS file system will ensure program compatibility with future MS-DOS versions through compatible disk formats and consistent internal storage. This will ensure compatibility with future MS-DOS versions.

Use file handles instead of FCBs.

A handle is a 16-bit number that is returned by MS-DOS when a file is opened or created using Functions 3CH, 3DH, 5AH, or 5BH (Create Handle, Open Handle, Create Temporary File, or Create New File). The MS-DOS file-related function requests that use handles are listed in Table 1.5 in Chapter 1, "System Calls."

These calls should be used instead of the old file-related functions that use FCBs (file control blocks). This is because a file operation can simply pass its handle rather than having to maintain FCB information. If FCBs must be used, be sure the program closes them and does not move them around in memory.

Close all files that have changed in length before issuing an Interrupt 20H (Program Terminate), Function 00H (Terminate Program), Function 4CH (End Process), or Function 0DH (Reset Disk).

If a changed file is not closed, its length will not be recorded correctly in the directory.

Close all files when they are no longer needed.

Closing unneeded files will optimize performance in a networking environment.

Only change disks if all files on the disk are closed.

Information in internal system buffers may be written incorrectly to a changed disk.

7.7.1 Locking Files

Programs should not rely on being denied access to a locked region.

Determine the status of the region by attempting to lock it, and examine the error code.

Programs should not close a file with a locked region or terminate with an open file that contains a locked region.

The result is undefined. Programs that might be terminated by an Interrupt 23H or Interrupt 24H (Control-C Handler Address or Critical Error Handler Address) should trap these interrupts and unlock any locked regions before exiting.

7.8 MISCELLANEOUS

Avoid timing dependencies.

Various machines use CPUs of different speeds. Also, programs that rely upon the speed of the clock for timing will not be dependable in a networking environment.

Use the documented interface to the operating system. If either the hardware or media change, the operating system will be able to use the features without modification.

Don't use the OEM (Original Equipment Manufacturer) -provided ROM support.

Don't directly address the video memory.

Don't use undocumented function calls, interrupts, or features. These items may change or not continue to exist in future versions of MS-DOS. Use of these features would make your program highly non-portable.

Use the .EXE format rather than the .COM format.

.EXE files are relocatable and .COM files are direct memory images that load at a specific place and have no room for additional control information to be placed in them. .EXE files have headers that can be expanded for compatibility with future versions of MS-DOS.

Use the environment to pass information to applications.

The environment allows a parent process to pass information to a child process. COMMAND.COM is usually the parent process to every application, so default drive and path information can easily be passed to the application.

INDEX