MICROSOFT™
MS-DOS™
Operating System

Programmer's Reference

MICROSOFT™ MS-DOS™ Programmer's Reference

Microsoft® MS™-DOS

Operating System

Programmer's Reference Manual

under the licence from Microsoft Corporation
Printed in Taiwan

Microsoft Corporation

Comments about this documentation may be sent to:

Microsoft Corporation
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

Document No. 8411-200-00
Part No. 14F36PM

under the licence from Microsoft Corporation
Printed in Taiwan

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

GENERAL INTRODUCTION

The Microsoft(R) MS(tm)-DOS Programmer's Reference Manual is a technical reference manual for system programmers. This manual contains a description and examples of all MS-DOS 2.0 system calls and interrupts (Chapter 1). Chapter 2, "MS-DOS 2.0 Device Drivers" contains information on how to install your own device drivers on MS-DOS. Two examples of device driver programs (one serial and one block) are included in Chapter 2. Chapters 3 through 5 contain technical information about MS-DOS, including MS-DOS disk allocation (Chapter 3), MS-DOS control blocks and work areas (Chapter 4), and EXE file structure and loading (Chapter 5).

CHAPTER 1

SYSTEM CALLS

1.1 INTRODUCTION

MS-DOS provides two types of system calls: interrupts and function requests. This chapter describes the environments from which these routines can be called, how to call them, and the processing performed by each.

1.2 PROGRAMMING CONSIDERATIONS

The system calls mean you don't have to invent your own ways to perform these primitive functions, and make it easier to write machine-independent programs.

1.2.1 Calling From Macro Assembler

The system calls can be invoked from Macro Assembler simply by moving any required data into registers and issuing an interrupt. Some of the calls destroy registers, so you may have to save registers before using a system call. The system calls can be used in macros and procedures to make your programs more readable; this technique is used to show examples of the calls.

1.2.2 Calling From A High-Level Language

The system calls can be invoked from any high-level language whose modules can be linked with assembly-language modules.

<u>Calling from Microsoft Basic</u>: Different techniques are used to invoke system calls from the compiler and interpreter. Compiled modules can be linked with assembly-language modules; from the interpreter, the CALL statement or USER function can be used to execute the appropriate 8086 object code.

Calling from Microsoft Pascal: In addition to linking with an assembly-language module, Microsoft Pascal includes a function (DOSXQQ) that can be used directly from a Pascal program to call a function request.

Calling from Microsoft FORTRAN: Modules compiled with Microsoft FORTRAN can be linked with assembly-language modules.

1.2.3 Returning Control To MS-DOS

Control can be returned to MS-DOS in any of four ways:

1. Call Function Request 4CH

   ```
   MOV  AH,4CH
   INT  21H
   ```

   This is the preferred method.

2. Call Interrupt 20H:

   ```
   INT  20H
   ```

3. Jump to location 0 (the beginning of the Program Segment Prefix):

   ```
   JMP  0
   ```

   Location 0 of the Program Segment Prefix contains an INT 20H instruction, so this technique is simply one step removed from the first.

4. Call Function Request 00H:

   ```
   MOV  AH,00H
   INT  21H
   ```

   This causes a jump to location 0, so it is simply one step removed from technique 2, or two steps removed from technique 1.

1.2.4 Console And Printer Input/Output Calls

The console and printer system calls let you read from and write to the console device and print on the printer without using any machine-specific codes. You can still take advantage of specific capabilities (display attributes such as positioning the cursor or erasing the screen, printer attributes such as double-strike or underline, etc.) by using constants for these codes and reassembling once with the correct constant values for the attributes.

1.2.5 Disk I/O System Calls

Many of the system calls that perform disk input and output require placing values into or reading values from two system control blocks: the File Control Block (FCB) and directory entry.

1.3 FILE CONTROL BLOCK (FCB)

The Program Segment Prefix includes room for two FCBs at offsets 5CH and 6CH. The system call descriptions refer to unopened and opened FCBs. An <u>unopened</u> FCB is one that contains only a drive specifier and filename, which can contain wild card characters (* and ?). An <u>opened</u> FCB contains all fields filled by the Open File system call (Function 0FH). Table 1.1 describes the fields of the FCB.

Table 1.1 Fields of File Control Block (FCB)

Name	Size (bytes)	Offset Hex	Offset Decimal
Drive number	1	00H	0
Filename	8	01-08H	1-8
Extension	3	09-0BH	9-11
Current block	2	0CH,0DH	12,13
Record size	2	0EH,0FH	14,15
File size	4	10-13H	16-19
Date of last write	2	14H,15H	20,21
Time of last write	2	16H,17H	22,23
Reserved	8	18-1FH	24-31
Current record	1	20H	32
Relative record	4	21-24H	33-36

1.3.1 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 to be used to create or open a file, this field can be set to 0 to specify the default drive; the Open File system call Function (0FH) 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 LPT1), 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
| Y | Y | Y | Y | Y | Y | Y | M |
 15                        9 8
```

```
Offset 14H
| M | M | M | D | D | D | D | D |
          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
| H | H | H | H | H | M | M | M |
 15              11 10
```

```
Offset 16H
| M | M | M | S | S | S | S | S |
          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 Address.

1.3.2 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 (Decimal)
Flag byte (255, or FFH)	1	-7
Reserved	5	-6
Attribute byte: 02H = Hidden file 04H = System file	1	-1

1.3.3 Directory Entry

A directory contains one entry for each file on the disk. Each entry is 32 bytes; Table 1.2 describes the fields of an entry.

Table 1.2 Fields of Directory Entry

Name	Size (bytes)	Offset Hex	Offset Decimal
Filename	8	00-07H	0-7
Extension	3	08-0AH	8-10
Attributes	1	0BH	11
Reserved	10	0C-15H	12-21

Time of last write	2	16H,17H	22,23
Date of last read	2	18H,19H	24,25
Reserved	2	1AH,1BH	26,27
File size	4	1C-1FH	28-31

1.3.4 Fields Of The FCB

Filename (offset 00H): Eight characters, left-aligned and padded (if necessary) with blanks. MS-DOS uses the first byte of this field for two special codes:

00H (0)	End of allocated directory
E5H (229)	Free directory entry

Extension (offset 08H): Three characters, left-aligned and padded (if necessary) with blanks. This field can be all blanks (no extension).

Attributes (offset 0BH): Attributes of the file:

Value Hex	Binary	Dec	Meaning
01H	0000 0001	1	Read-only
02H	0000 0010	2	Hidden
04H	0000 0100	4	System
07H	0000 0111	7	Changeable with CHGMOD
08H	0000 1000	8	Volume-ID
0AH	0001 0000	10	Directory
16H	0001 0110	22	Hard attributes for FINDENTRY
20H	0020 0000	32	Archive

Reserved (offset 0CH): Reserved for MS-DOS.

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
| H | H | H | H | H | M | M | M |
 15                11 10
```

```
Offset 16H
| M | M | M | S | S | S | S | S |
          5 4                 0
```

Date of Last Write (offset 18H): The 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 |
 15                        9 8

Offset 18H
| M | M | M | D | D | D | D | D |
           5 4                0
```

File Size (offset 1CH): The size of the file, in bytes. The first word of this 4-byte field is the low-order part of the size.

1.4 SYSTEM CALL DESCRIPTIONS

Many system calls require that parameters be loaded into one or more registers before the call is issued; most calls return information in the registers (usually a code that describes the success or failure of the operation). The description of system calls 00H-2EH 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 its use.

The description of system calls 2FH-57H includes the following:

A drawing of the 8088 registers that shows their contents before and after the system call.

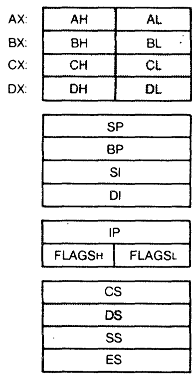
A more complete description of the register contents required before the system call.

A description of the processing performed.

Error returns from the system call.

An example of its use.

Figure 1 is an example of how each system call is described. Function 27H, Random Block Read, is shown.



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

Return
AL
  0 = Read completed successfully
  1 = EOF
  2 = End of segment
  3 = EOF, partial record
CX
  Number of blocks read

Figure 1. Example of System Call Description

1.4.1 Programming Examples

A macro is defined for each system call, then used in some examples. In addition, a few other macros are defined for use in the examples. The use of macros allows the examples to be more complete programs, rather than isolated uses of the system calls. All macro definitions are listed at the end of the chapter.

The examples are not intended to represent good programming practice. In particular, error checking and good human interface design have been sacrificed to conserve space. You may, however, find the macros a convenient way to include system calls in your assembly language programs.

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 hex.

1.5 XENIX COMPATIBLE CALLS

MS-DOS 2.0 supports hierarchical (i.e., tree-structured) directories, similar to those found in the Xenix operating system. (For information on tree-structured directories, refer to the MS-DOS User's Guide.)

The following system calls are compatible with the Xenix system:

Function 39H	Create Sub-Directory
Function 3AH	Remove a Directory Entry
Function 3BH	Change the Current Directory
Function 3CH	Create a File
Function 3DH	Open a File
Function 3FH	Read From File/Device
Function 40H	Write to a File or Device
Function 41H	Delete a Directory Entry
Function 42H	Move a File Pointer
Function 43H	Change Attributes
Function 44H	I/O Control for Devices
Function 45H	Duplicate a File Handle
Function 46H	Force a Duplicate of a Handle
Function 4BH	Load and Execute a Program
Function 4CH	Terminate a Process
Function 4DH	Retrieve Return Code of a Child

There is no restriction in MS-DOS 2.0 on the depth of a tree (the length of the longest path from root to leaf) except in the number of allocation units available. The root directory will have a fixed number of entries (64 for the single-sided disk). For non-root directories, the number of files per directory is only limited by the number of allocation units available.

Pre-2.0 disks will appear to MS-DOS 2.0 as having only a root directory with files in it and no subdirectories.

Implementation of the tree structure is simple. The root directory is the pre-2.0 directory. Subdirectories of the root have a special attribute set indicating that they are directories. The subdirectories themselves are files, linked through the FAT as usual. Their contents are identical in character to the contents of the root directory.

Pre-2.0 programs that use system calls not described in this chapter will be unable to make use of files in other directories. Those files not necessary for the current task will be placed in other directories.

Attributes apply to the tree-structured directories in the following manner:

Attribute	Meaning/Function for files	Meaning/Function for directories
volume_id	Present at the root. Only one file may have this set.	Meaningless.
directory	Meaningless.	Indicates that the directory entry is a directory. Cannot be changed with 43H.
read_only	Old fcb-create, new Create, new open (for write or read/write) will fail.	Meaningless.
archive	Set when file is written. Set/reset via Function 43H.	Meaningless.
hidden/ system	Prevents file from being found in search first/search next. Old open will fail.	Prevents directory entry from being found. Function 3BH will still work.

1.6 INTERRUPTS

MS-DOS reserves interrupts 20H through 3FH for its own use. The table of interrupt routine addresses (vectors) is maintained in locations 80H-FCH. Table 1.3 lists the interrupts in numeric order; Table 1.4 lists the interrupts in alphabetic order (of the description). User programs should only issue Interrupts 20H, 21H, 25H, 26H, and 27H. (Function Requests 4CH and 31H are the preferred method for Interrupts 20H and 27H for versions of MS-DOS that are 2.0 and higher.)

NOTE

Interrupts 22H, 23H, and 24H are not interrupts that can be issued by user programs; they are simply locations where a segment and offset address are stored.

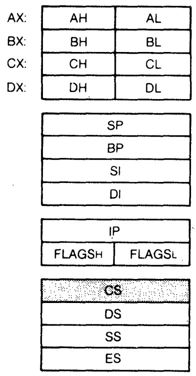
Table 1.3 MS-DOS Interrupts, Numeric Order

Interrupt Hex	Dec	Description
20H	32	Program Terminate
21H	33	Function Request
22H	34	Terminate Address
23H	35	<CTRL-C> Exit Address
24H	36	Fatal Error Abort Address
25H	37	Absolute Disk Read
26H	38	Absolute Disk Write
27H	39	Terminate But Stay Resident
28-40H	40-64	RESERVED -- DO NOT USE

Table 1.4 MS-DOS Interrupts, Alphabetic Order

Description	Interrupt Hex	Dec
Absolute Disk Read	25H	37
Absolute Disk Write	26H	38
<CTRL-C> Exit Address	23H	35
Fatal Error Abort Address	24H	36
Function Request	21H	33
Program Terminate	20H	32
RESERVED -- DO NOT USE	28-40H	40-64
Terminate Address	22H	34
Terminate But Stay Resident	27H	39

Program Terminate (Interrupt 20H)



Call
CS
  Segment address of Program Segment Prefix

Return
None

Interrupt 20H causes the current process to terminate and returns control to its parent process. All open file handles are closed and the disk cache is cleaned. This interrupt is almost always is used in old .COM files for termination.

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 Program Segment Prefix:

Exit Address	Offset
Program Terminate	0AH
CONTROL-C	0EH
Critical Error	12H

All file buffers are flushed to disk.

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.

Interrupt 20H is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function Request 4CH, Terminate a Process.

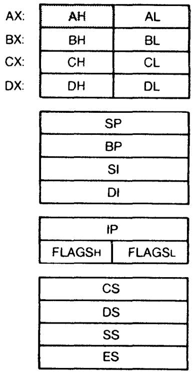
Macro Definition: terminate macro

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

Example

```
;CS must be equal to PSP values given at program start
;(ES and DS values)
   INT 20H
;There is no return from this interrupt
```

Function Request (Interrupt 21H)



Call
AH
  Function number
Other registers as specified in
  individual function

Return
As specified in individual function

The AH register must contain the number of the system function. See Section 1.7, "Function Requests," for a description of the MS-DOS system functions.

NOTE

No macro is defined for this interrupt, because all function descriptions in this chapter that define a macro include Interrupt 21H.

Example

To call the Get Time function:

```
mov    ah,2CH        ;Get Time is Function 2CH
int    21H           ;THIS INTERRUPT
```

Terminate Address (Interrupt 22H)
CONTROL-C Exit Address (Interrupt 23H)
Fatal Error Abort Address (Interrupt 24H)

These are not true interrupts, but rather storage locations for a segment and offset address. The interrupts are issued by MS-DOS under the specified circumstance. You can change any of these addresses with Function Request 25H (Set Vector) if you prefer to write your own interrupt handlers.

Interrupt 22H -- Terminate Address

When a program terminates, control transfers to the address at offset 0AH of the Program Segment Prefix. This address is copied into the Program Segment Prefix, from the Interrupt 22H vector, when the segment is created.

Interrupt 23H -- CONTROL-C Exit Address

If the user types CONTROL-C during keyboard input or display output, control transfers to the INT 23H vector in the interrupt table. This address is copied into the Program Segment Prefix, from the Interrupt 23H vector, when the segment is created.

If the CONTROL-C routine preserves all registers, it can end with an IRET instruction (return from interrupt) to continue program execution. 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 -- so 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 (ETX-CR-LF) is sent to the display and the function resumes at the beginning of the next line.

If the program creates a new segment and loads a second program that changes the CONTROL-C address, termination of the second program restores the CONTROL-C address to its value before execution of the second program.

Interrupt 24H -- Fatal Error Abort Address

If a fatal disk error occurs during execution of one of the disk I/O function calls, control transfers to the INT 24H vector in the vector table. This address is copied into the Program Segment Prefix, from the Interrupt 24H vector, when the segment is created.

BP:SI contains the address of a Device Header Control Block from which additional information can be retrieved.

NOTE

Interrupt 24H is not issued if the failure occurs during execution of Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write). These errors are usually handled by the MS-DOS 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 give you the information you need about interpreting the error codes, managing the registers and stack, and controlling the system's response to the error in order to write your own error-handling routines.

Error Codes

When an error-handling program gains control from Interrupt 24H, the AX and DI registers can contain codes that describe the error. If Bit 7 of AH is 1, the error is either a bad image of the File Allocation Table or an error occurred on a character device. The device header passed in BP:SI can be examined to determine which case exists. If the attribute byte high order bit indicates a block device, then the error was a bad FAT. Otherwise, the error is on a character device.

The following are error codes for Interrupt 24H:

Error Code	Description
0	Attempt to write on write-protected disk
1	Unknown unit
2	Drive not ready
3	Unknown command
4	Data error
5	Bad 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

The user stack will be in effect (the first item described below is at the top of the stack), and will contain the following from top to bottom:

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

AX      User registers at time of original
BX      INT 21H request
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 an IRET is executed, MS-DOS will respond according to (AL) as follows:

```
(AL)=0   ignore the error
    =1   retry the operation
    =2   terminate the program via INT 23H
```

Notes:

1. Before giving this routine control for disk errors, MS-DOS performs five retries.

2. For disk errors, this exit is taken only for errors occurring during an Interrupt 21H. It is not used for errors during Interrupts 25H or 26H.

3. This routine is entered in a disabled state.

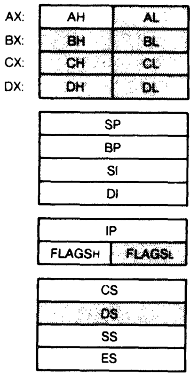
4. The SS, SP, DS, ES, BX, CX, and DX registers must be preserved.

5. This interrupt handler should refrain from using MS-DOS funtion calls. If necessary, it may use calls 01H through 0CH. Use of any other call will destroy the MS-DOS stack and will leave MS-DOS in an unpredictable state.

6. The interrupt handler must not change the contents of the device header.

7. If the interrupt handler will handle errors rather than returning to MS-DOS, it should restore the application program's registers from the stack, remove all but the last three words on the stack, then issue an IRET. This will return to the program immediately after the INT 21H that experienced the error. Note that if this is done, MS-DOS will be in an unstable state until a function call higher than 0CH is 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.

This interrupt transfers control to the MS-DOS BIOS. 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.

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. (This is necessary because data is passed back in the flags.) 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 meaning).

Macro Definition:

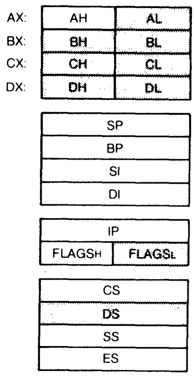
```
abs_disk_read  macro   disk,buffer,num_sectors,start
               mov     al,disk
               mov     bx,offset buffer
               mov     cx,num_sectors
               mov     dh,start
               int     25H
               endm
```

Example

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

```
prompt      db    "Source in A, target in B",13,10
            db    "Any key to start. $"
start       dw     0
buffer      db     64 dup (512 dup (?))   ;64 sectors
            .
            .
int_25H:    display prompt            ;see Function 09H
            read_kbd                  ;see Function 08H
            mov      cx,5             ;copy 5 groups of
                                        ;64 sectors
copy:       push     cx               ;save the loop counter
            abs_disk_read  0,buffer,64,start  ;THIS INTERRUPT
            abs_disk_write 1,buffer,64,start  ;see INT 26H
            add  start,64             ;do the next 64 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

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 the MS-DOS BIOS. 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.

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. (This is necessary because data is passed back in the flags.) 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 meaning).

Macro Definition:

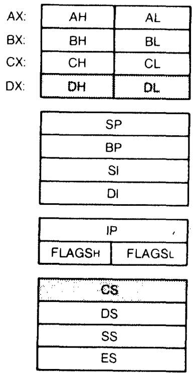
```
abs_disk write  macro  disk,buffer,num_sectors,start
                mov    al,disk
                mov    bx,offset buffer
                mov    cx,num_sectors
                mov    dh,start
                int    26H
                endm
```

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          equ    0
on           equ    1
             .
             .
prompt       db     "Source in A, target in B",13,10
             db     "Any key to start. $"
start        dw     0
buffer       db     64 dup (512 dup (?))  ;64 sectors
             .
             .
int_26H:     display prompt         ;see Function 09H
             read_kbd               ;see Function 08H
             verify  on             ;see Function 2EH
             mov     cx,5           ;copy 5 groups of 64 sectors
copy:        push    cx             ;save the loop counter
             abs_disk_read  0,buffer,64,start  ;see INT 25H
             abs_disk_write 1,buffer,64,start  ;THIS INTERRUPT
             add  start,64          ;do the next 64 sectors
             pop  cx                ;restore the loop counter
             loop copy
             verify  off            ;see Function 2EH
```

Terminate But Stay Resident (Interrupt 27H)



Call
CS:DX
  First byte following
  last byte of code

Return
None

The Terminate But Stay Resident call is used to make a piece of code remain resident in the system after its termination. Typically, this call is used in .COM files to allow some device-specific interrupt handler to remain resident to process asynchronous interrupts.

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 but is treated as an extension of MS-DOS; it remains resident and is not overlaid by other programs when it terminates.

This interrupt is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function 31H, Keep Process.

Macro Definition:

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

Example

```
;CS must be equal to PSP values given at program start
;(ES and DS values)
  mov     DX,LastAddress
  int    27H
;There is no return from this interrupt
```

1.7 FUNCTION REQUESTS

Most of the MS-DOS function calls require input to be passed to them in registers. After setting the proper register values, the function may be invoked in one of the following ways:

1. Place the function number in AH and execute a long call to offset 50H in your Program Segment Prefix. Note that programs using this method will not operate correctly on versions of MS-DOS that are lower than 2.0.

2. Place the function number in AH and issue Interrupt 21H. All of the examples in this chapter use this method.

3. An additional method exists for programs that were written with different calling conventions. This method should be avoided for all new programs. The function number is placed in the CL register and other registers are set according to the function specification. Then, an intrasegment call is made to location 5 in the current code segment. That location contains a long call to the MS-DOS function dispatcher. Register AX is always destroyed if this method is used; otherwise, it is the same as normal function calls. Note that this method is valid only for Function Requests 00H through 024H.

1.7.1 CP/M(R)-Compatible Calling Sequence

A different sequence can be used for programs that must conform to CP/M calling conventions:

1. Move any required data into the appropriate registers (just as in the standard sequence).

2. Move the function number into the CL register.

3. Execute an intrasegment call to location 5 in the current code segment.

This method can only be used with functions 00H through 24H that do not pass a parameter in AL. Register AX is always destroyed when a function is called in this manner.

1.7.2 Treatment Of Registers

When MS-DOS takes control after a function call, 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.

IMPORTANT NOTE

The macro definitions and extended example for MS-DOS system calls 00H through 2EH can be found at the end of this chapter.

Table 1.5 lists the function requests in numeric order; Table 1.6 list the function requests in alphabetic order (of the description).

Table 1.5 MS-DOS Function Requests, Numeric Order

Function Number	Function Name
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	Disk Reset
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
19H	Current Disk
1AH	Set Disk Transfer Address
21H	Random Read

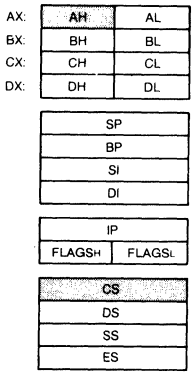
22H	Random Write
23H	File Size
24H	Set Relative Record
25H	Set Vector
27H	Random Block Read
28H	Random Block Write
29H	Parse File Name
2AH	Get Date
2BH	Set Date
2CH	Get Time
2DH	Set Time
2EH	Set/Reset Verify Flag
2FH	Get Disk Transfer Address
30H	Get DOS Version Number
31H	Keep Process
33H	CONTROL-C Check
35H	Get Interrupt Vector
36H	Get Disk Free Space
38H	Return Country-Dependent Information
39H	Create Sub-Directory
3AH	Remove a Directory Entry
3BH	Change Current Directory
3CH	Create a File
3DH	Open a File
3EH	Close a File Handle
3FH	Read From File/Device
40H	Write to a File/Device
41H	Delete a Directory Entry
42H	Move a File Pointer
43H	Change Attributes
44H	I/O Control for Devices
45H	Duplicate a File Handle
46H	Force a Duplicate of a Handle
47H	Return Text of Current Directory
48H	Allocate Memory
49H	Free Allocated Memory
4AH	Modify Allocated Memory Blocks
4BH	Load and Execute a Program
4CH	Terminate a Process
4DH	Retrieve the Return Code of a Child
4EH	Find Match File
4FH	Step Through a Directory Matching Files
54H	Return Current Setting of Verify
56H	Move a Directory Entry
57H	Get/Set Date/Time of File

Table 1.6 MS-DOS Function Requests, Alphabetic Order

Function Name	Number
Allocate Memory	48H
Auxiliary Input	03H
Auxiliary Output	04H
Buffered Keyboard Input	0AH
Change Attributes	43H
Change the Current Directory	3BH
Check Keyboard Status	0BH
Close a File Handle	3EH
Close File	10H
CONTROL-C Check	33H
Create a File	3CH
Create File	16H
Create Sub-Directory	39H
Current Disk	19H
Delete a Directory Entry	41H
Delete File	13H
Direct Console Input	07H
Direct Console I/O	06H
Disk Reset	0DH
Display Character	02H
Display String	09H
Duplicate a File Handle	45H
File Size	23H
Find Match File	4EH
Flush Buffer, Read Keyboard	0CH
Force a Duplicate of a Handle	46H
Free Allocated Memory	49H
Get Date	2AH
Get Disk Free Space	36H
Get Disk Transfer Address	2FH
Get DOS Version Number	30H
Get Interrupt Vector	35H
Get Time	2CH
Get/Set Date/Time of File	57H
I/O Control for Devices	44H
Keep Process	31H
Load and Execute a Program	4BH
Modify Allocated Memory Blocks	4AH
Move a Directory Entry	56H
Move a File Pointer	42H
Open a File	3DH
Open File	0FH
Parse File Name	29H
Print Character	05H
Random Block Read	27H
Random Block Write	28H
Random Read	21H
Random Write	22H
Read From File/Device	3FH
Read Keyboard	08H
Read Keyboard and Echo	01H

Remove a Directory Entry	3AH
Rename File	17H
Retrieve the Return Code of a Child	4DH
Return Current Setting of Verify	54H
Return Country-Dependent Information	38H
Return Text of Current Directory	47H
Search for First Entry	11H
Search for Next Entry	12H
Select Disk	0EH
Sequential Read	14H
Sequential Write	15H
Set Date	2BH
Set Disk Transfer Address	1AH
Set Relative Record	24H
Set Time	2DH
Set Vector	25H
Set/Reset Verify Flag	2EH
Step Through a Directory Matching	4FH
Terminate a Process	4CH
Terminate Program	00H
Write to a File/Device	40H

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:

Program terminate	0AH
CONTROL-C	0EH
Critical error	12H

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.

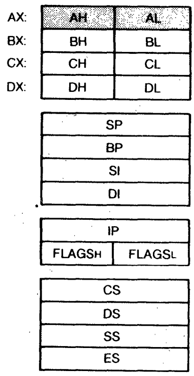
Macro Definition:

```
terminate_program  macro
                   xor    ah,ah
                   int    21H
                   endm
```

Example

```
;CS must be equal to PSP values given at program start
;(ES and DS values)
  mov   ah,0
  int   21H
;There are no returns from this interrupt
```

Read Keyboard and Echo (Function 01H)



Call
AH = 01H

Return
AL
  Character typed

Function 01H waits for a character to be typed at the keyboard, then echos the character to the display and returns it in AL. If the character is CONTROL-C, Interrupt 23H is executed.

Macro Definition:

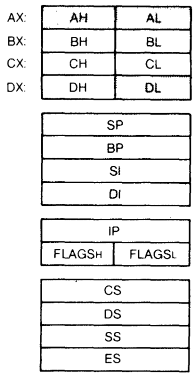
```
read_kbd_and_echo  macro
                   mov  ah, 01H
                   int  21H
                   endm
```

Example

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

```
func_01H:  read_kbd_and_echo          ;THIS FUNCTION
           print_char    al           ;see Function 05H
           cmp           al,0DH       ;is it a CR?
           jne           func_01H     ;no, print it
           print_char    10           ;see Function 05H
           display_char  10           ;see Function 02H
           jmp           func_01H     ;get another character
```

Display Character (Function 02H)



Call
AH = 02H
DL
  Character to be displayed

Return
None

Function 02H displays the character in DL. If CONTROL-C is typed, Interrupt 23H is issued.

Macro Definition:

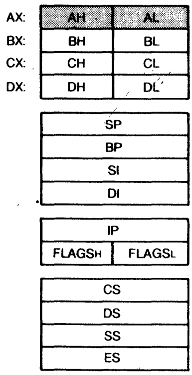
```
display_char macro   character
             mov     dl,character
             mov     ah,02H
             int     21H
             endm
```

Example

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

```
func_02H:  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     func_02H:        ;get another character
```

Auxiliary Input (Function 03H)



Call
AH = 03H

Return
AL
  Character from auxiliary device

Function 03H waits for a character from the auxiliary input device, 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:

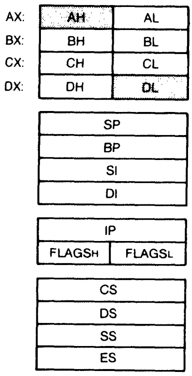
```
aux_input  macro
           mov   ah,03H
           int   21H
           endm
```

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:

```
func_03H:  aux_input         ;THIS FUNCTION
           cmp   al,1AH      ;end of file?
           je    continue    ;yes, all done
           print_char  al    ;see Function 05H
           jmp   func_03H    ;get another character
continue:  .
           .
```

Auxiliary Output (Function 04H)



Call
AH = 04H
DL
  Character for auxiliary device

Return
None

Function 04H sends the character in DL to the auxiliary output device. 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:

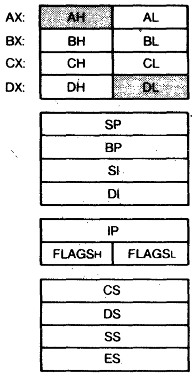
```
aux_output  macro  character
            mov   dl,character
            mov   ah,04H
            int   21H
            endm
```

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
         .
         .
func_04H:get_string  80,string        ;see Function 0AH
         cmp  string[1],0             ;null string?
         je   continue                ;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  func_04H                ;get another string
continue: .
          .
```

Print Character (Function 05H)



Call
AH = 05H
DL
  Character for printer

Return
None

Function 05H prints the character in DL on the standard printer device. If CONTROL-C has been typed at console input, Interrupt 23H is issued.

Macro Definition:

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

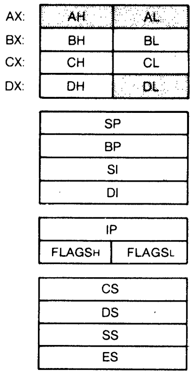
Example

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

```
line_num    db    0
            .
            .
func_05H:   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
            mov   bl,33          ;start over with (!)
```

```
no_reset:   loop  print_it        ;print another character
            print_char 13         ;carriage return
            print_char 10         ;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 (255) before call, then Zero flag set means AL has character from keyboard.
  Zero flag not set means there was not a character to get, and AL = 0

The processing depends on the value in DL when the function is called:

> DL is FFH (255) -- If a character has been typed at the keyboard, it is returned in AL and the Zero flag is 0; if a character has not been typed, the Zero flag is 1.
>
> DL is not FFH -- The character in DL is displayed.

This function does <u>not</u> check for CONTROL-C.

Macro Definition: dir_console_io

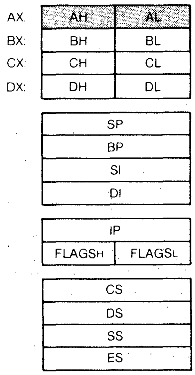
```
dir_console_io  macro switch
                mov   dl,switch
                mov   ah,06H
                int   21H
                endm
```

Example

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

```
time            db  "00:00:00.00",13,10,"$"  ;see Function 09H
;                                           ;for explanation of $
ten             db  10
                .
                .
func_06H:       set_time  0,0,0,0          ;see Function 2DH
read_clock:     get_time                   ;see Function 2CH
                convert  ch,ten,time       ;see end of chapter
                convert  cl,ten,time[3]    ;see end of chapter
                convert  dh,ten,time[6]    ;see end of chapter
                convert  dl,ten,time[9]    ;see end of chapter
                display  time              ;see Function 09H
                dir_console_io  FFH        ;THIS FUNCTION
                jne      stop              ;yes, stop timer
                jmp      read_clock        ;no, keep timer
                                           ;running
stop:           read_kbd                   ;see Function 08H
                jmp      func_06H          ;start over
```

Direct Console Input (Function 07H)



Call
AH = 07H

Return
AL
  Character from keyboard

Function 07H waits for a character to be typed, 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 Functions 01H or 08H.)

Macro Definition:

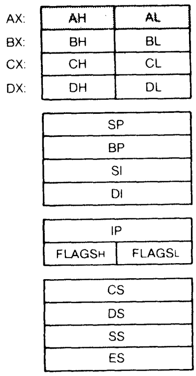
```
dir_console_input  macro
                   mov   ah,07H
                   int   21H
                   endm
```

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 $
          .
          .
func_07H: 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   continue             ;yes, all done
          mov  password[bx],al      ;no, put character in string
          inc  bx                   ;bump index
          loop get_pass             ;get another character
continue: .                         ;BX has length of password+1
          .
```

Read Keyboard (Function 08H)



Call
AH = 08H

Return
AL
  Character from keyboard

Function 08H waits for a character to be typed, 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:

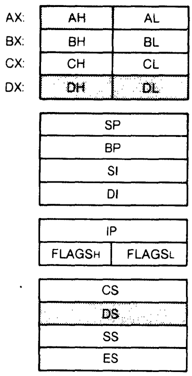
```
read_kbd  macro
          mov   ah,08H
          int   21H
          endm
```

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 $
          .
          .
func_08H: 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   continue         ;yes, all done
          mov  password[bx],al  ;no, put char. in string
          inc  bx               ;bump index
          loop get_pass         ;get another character
continue: .                     ;BX has length of password+1
          .
```

Display String (Function 09H)



Call
AH = 09H
DS:DX
  String to be displayed

Return
None

DX must contain the offset (from the segment address in DS) of a string that ends with "$". The string is displayed (the $ is not displayed).

Macro Definition:

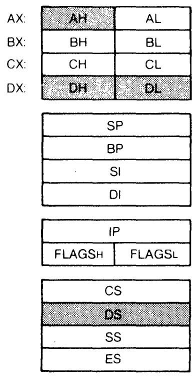
```
display  macro string
         mov   dx,offset string
         mov   ah,09H
         int   21H
         endm
```

Example

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

```
table    db     "0123456789ABCDEF"
sixteen  db      16
result   db     " - 00H",13,10,"$"   ;see text for
                                     ;explanation of $
         .
         .
func_09H:read_kbd_and_echo           ;see Function 01H
         convert al,sixteen,result[3] ;see end of chapter
         display result              ;THIS FUNCTION
         jmp     func_09H            ;do it again
```

Buffered Keyboard Input (Function 0AH)



Call
AH = 0AH
DS:DX
  Input buffer

Return
None

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 CR (you must set this value).
2	Actual number of characters typed, not counting the CR (the function sets this value).
3-n	Buffer; must be at least as long as the number in byte 1.

This function waits for characters to be typed. Characters are read from the keyboard and placed in the buffer beginning at the third byte until RETURN is typed. If the buffer fills to one less than the maximum, additional characters typed are ignored and ASCII 7 (BEL) is sent to the display until RETURN is pressed. The string can be edited as it is being entered. If CONTROL-C is typed, Interrupt 23H is issued.

The second byte of the buffer is set to the number of characters entered (not counting the CR).

Macro Definition:

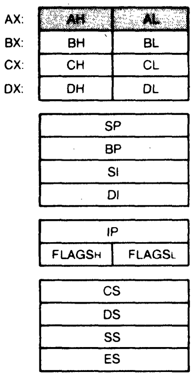
```
get_string  macro  limit,string
            mov    dx,offset string
            mov    string,limit
            mov    ah,0AH
            int    21H
            endm
```

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    13,10,"$"
                  .
                  .
func 0AH:         get_string 17,buffer     ;THIS FUNCTION
                  xor  bx,bx               ;so byte can be
                                           ;used as index
                  mov  bl,chars_entered    ;get string length
                  mov  buffer[bx+2],"$"    ;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
  255 (FFH) = characters in type-ahead buffer
  0 = no characters in type-ahead buffer

Checks whether there are characters in the type-ahead buffer. If so, AL returns FFH (255); if not, AL returns 0. If CONTROL-C is in the buffer, Interrupt 23H is executed.

Macro Definition:

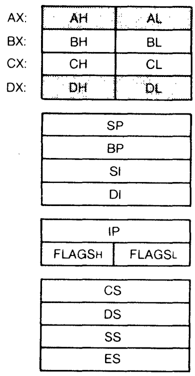
```
check_kbd_status  macro
                  mov    ah,0BH
                  int    21H
                  endm
```

Example

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

```
time        db      "00:00:00.00",13,10,"$"
ten         db       10
            .
            .
func_0BH:   get_time                        ;see Function 2CH
            convert ch,ten,time             ;see end of chapter
            convert cl,ten,time[3]          ;see end of chapter
            convert dh,ten,time[6]          ;see end of chapter
            convert dl,ten,time[9]          ;see end of chapter
            display time                    ;see Function 09H
            check_kbd_status                ;THIS FUNCTION
            cmp     al,FFH                  ;has a key been typed?
            je      all_done                ;yes, go home
            jmp     func_0BH                ;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.

The keyboard type-ahead buffer is emptied. 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:

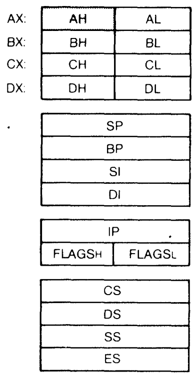
```
flush_and_read_kbd  macro switch
                    mov   al,switch
                    mov   ah,0CH
                    int   21H
                    endm
```

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.

```
func_0CH: flush_and_read_kbd 1      ;THIS FUNCTION
          print_char   al           ;see Function 05H
          cmp          al,0DH       ;is it a CR?
          jne          func_0CH     ;no, print it
          print_char   10           ;see Function 05H
          display_char 10           ;see Function 02H
          jmp          func_0CH     ;get another character
```

Disk Reset (Function 0DH)



Call
AH = 0DH

Return
None

Function 0DH is used to ensure that the internal buffer cache matches the disks in the drives. This function writes out dirty buffers (buffers that have been modified), and marks all buffers in the internal cache as free.

Function 0DH flushes all file buffers. It does not update directory entries; you must close files that have changed to update their directory entries (see Function 10H, Close File). This function need not be called before a disk change if all files that changed were closed. It is generally used to force a known state of the system; CONTROL-C interrupt handlers should call this function.

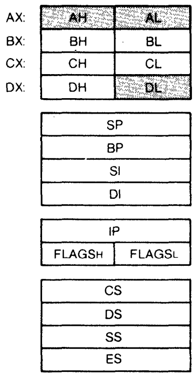
Macro Definition:

```
disk_reset macro disk
           mov   ah,0DH
           int   21H
           endm
```

Example

```
  mov     ah,0DH
  int     21H
;There are no errors returned by this call.
```

Select Disk (Function 0EH)



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

Return
AL
  Number of logical drives

The drive specified in DL (0 = A:, 1 = B:, etc.) is selected as the default disk. The number of drives is returned in AL.

Macro Definition:

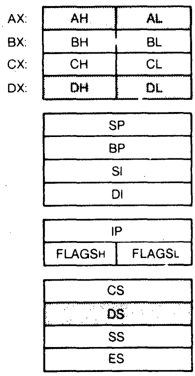
```
select_disk macro disk
            mov   dl,disk[-64]
            mov   ah,0EH
            int   21H
            endm
```

Example

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

```
func_0EH: current_disk        ;see Function 19H
          cmp     al,00H      ;drive A: selected?
          je      select_b    ;yes, select B
          select_disk "A"     ;THIS FUNCTION
          jmp     continue
select_b: select_disk "B"     ;THIS FUNCTION
continue: .
          .
```

Open File (Function 0FH)



Call
AH = 0FH
DS:DX
  Unopened FCB

Return
AL
  0 = Directory entry found
  255 (FFH) = No directory entry found

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 (default disk), it is changed to the actual disk used (1 = A:, 2 = B:, etc.). This lets you change the default disk without interfering with subsequent operations on this file.
>
> The Current Block field (offset 0CH) is set to zero.
>
> The Record Size (offset 0EH) is set to the system default of 128.
>
> The 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, AL returns FFH (255).

Macro Definition:

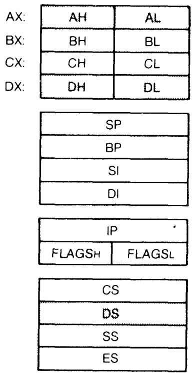
```
open    macro fcb
        mov   dx,offset fcb
        mov   ah,0FH
        int   21H
        endm
```

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      25 dup (?)
buffer          db      128 dup (?)
                .
                .
func_0FH:       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,128           ;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,128           ;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
  Opened FCB

Return
AL
  0 = Directory entry found
  FFH (255) = No directory entry found

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. This function must be called after a file is changed to update the directory entry.

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.

If a directory entry for the file is not found, AL returns FFH (255).

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

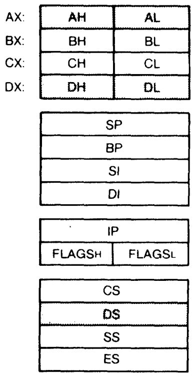
Example

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",13,10,"$"
fcb             db     2,"MOD1    BAS"
                db     25 dup (?)
buffer          db     128 dup (?)
                .
                .
func_10H:       set_dta  buffer          ;see Function 1AH
                open  fcb                ;see Function 0FH
                read_seq  fcb            ;see Function 14H
```

```
                cmp   buffer,FFH        ;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
  Unopened FCB

Return
  0 = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (from the segment address in DS) of an unopened FCB. The disk directory is searched for the first matching name. The name can have the ? wild card character to match any character. To search for hidden or system files, DX must point to the first byte of the extended FCB prefix.

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.

If a directory entry for the filename in the FCB is not found, AL returns FFH (255).

Notes:

If an extended FCB is used, the following search pattern is used:

1. If the FCB attribute is zero, only normal file entries are found. Entries for volume label, sub-directories, hidden, and system files will not be returned.

2. If the attribute field is set for hidden or system files, or directory entries, it is to be considered as an inclusive search. All normal file entries plus all entries matching the specified attributes are returned. To look at all directory entries except the volume label, the attribute byte may be set to hidden + system + directory (all 3 bits on).

3. If the attribute field is set for the volume label, it is considered an exclusive search, and only the volume label entry is returned.

Macro Definition:

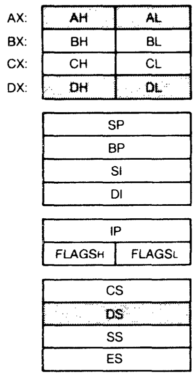
```
search_first macro fcb
             mov    dx,offset fcb
             mov    ah,11H
             int    21H
             endm
```

Example

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

```
yes         db    "FILE EXISTS.$"
no          db    "FILE DOES NOT EXIST.$"
fcb         db     2,"REPORT  ASM"
            db     25 dup (?)
buffer      db     128 dup (?)
            .
            .
func_11H:   set_dta   buffer              ;see Function 1AH
            search_first fcb              ;THIS FUNCTION
            cmp       al,FFH              ;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
  Unopened FCB

Return
AL
  0 = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (from the segment address in DS) of an FCB previously specified in a call to Function 11H. Function 12H is used after Function 11H (Search for First Entry) to find additional directory entries that match a filename that contains wild card characters. The disk directory is searched for the next matching name. The name can have the ? wild card character to match any character. To search for hidden or system files, DX must point to the first byte of the extended FCB prefix.

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.

If a directory entry for the filename in the FCB is not found, AL returns FFH (255).

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

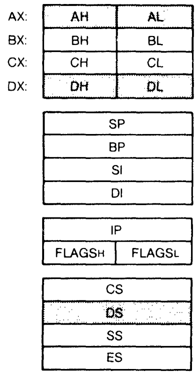
Example

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

```
message     db    "No files",10,13,"$"
files       db     0
ten         db     10
fcb         db     2,"???????????"
            db     25  dup (?)
buffer      db     128 dup (?)
```

```
             .
             .
func_12H:    set_dta  buffer          ;see Function 1AH
             search_first  fcb        ;see Function 11H
             cmp  al,FFH              ;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,FFH              ;directory entry found?
             je   done                ;no
             inc  files               ;yes, increment file
                                      ;counter
             jmp  search_dir          ;check again
done:        convert files,ten,message ;see end of chapter
all_done:    display  message         ;see Function 09H
```

Delete File (Function 13H)



Call
AH = 13H
DS:DX
  Unopened FCB

Return
  0 = Directory entry found
  FFH (255) = No directory entry found

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 the ? wild card character to match any character.

If a matching directory entry is found, it is deleted from the directory. If the ? wild card character is used in the filename, all matching directory entries are deleted. AL returns 0.

If no matching directory entry is found, AL returns FFH (255).

Macro Definition:

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

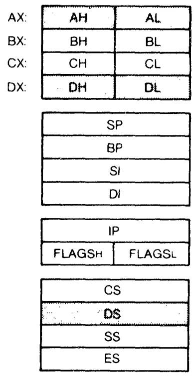
Example

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
ten             db      10
message         db     "NO FILES DELETED.",13,10,"$"
                                ;see Function 09H for
                                ;explanation of $
fcb             db      2,"???????????"
                db      25 dup  (?)
```

```
buffer          db    128 dup (?)
                .
                .
func_13H:       set_dta  buffer         ;see Function 1AH
                search_first fcb        ;see Function 11H
                cmp  al,FFH             ;directory entry found?
                je   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,ten,message ;see end of chapter
all_done:       display message         ;see Function 09H
```

Sequential Read (Function 14H)



Call
AH = 14H
DS:DX
  Opened FCB

Return
AL
  0 = Read completed successfully
  1 = EOF
  2 = DTA too small
  3 = EOF, partial record

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

The record size is set to the value at offset 0EH in 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:

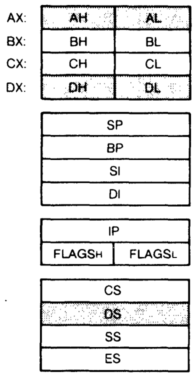
```
read_seq macro fcb
         mov   dx,offset fcb
         mov   ah,14H
         int   21H
         endm
```

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 26, or CONTROL-Z):

```
fcb             db     2,"TEXTFILEASC"
                db     25  dup (?)
buffer          db     128 dup (?),"$"
                .
                .
func_14H:       set_dta  buffer      ;see Function 1AH
                open  fcb            ;see Function 0FH
read_line:      read_seq  fc         ;THIS FUNCTION
                cmp    al,02H        ;end-of-file?
                je     all_done      ;yes
                cmp    al,02H        ;end-of-file with partial
                                     ;record?
                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 to next
                                     ;character
                jmp    find_eof      ;check next character
all_done:       close fcb            ;see Function 10H
```

Sequential Write (Function 15H)



Call
AH = 15H
DS:DX
  Opened FCB

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

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

The record size is set to the value at offset 0EH in 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	Transfer completed successfully.
1	Disk full; write canceled.
2	Not enough room at the Disk Transfer Address to write one record; write canceled

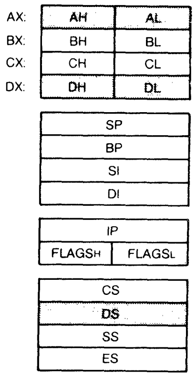
Macro Definition:
```
write_seq macro fcb
          mov   dx,offset fcb
          mov   ah,15H
          int   21H
          endm
```

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   14               ;offset of Record Size
                                       ;field in FCB
                .
                .
fcb1            db    2,"DIR     TMP"
                db    25 dup (?)
fcb2            db    2,"???????????"
                db    25 dup (?)
buffer          db    128 dup (?)
                .
                .
func_15H:   set_dta       buffer      ;see Function 1AH
            search_first  fcb2        ;see Function 11H
            cmp           al,FFH      ;directory entry found?
            je            all_done    ;no, no files on disk
            create        fcb1        ;see Function 16H
            mov           fcb1[record_size],12
                                      ;set record size to 12
write_it:   write_seq     fcb1        ;THIS FUNCTION
            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
```

Create File (Function 16H)



Call
AH = 16H
DS:DX
  Unopened FCB

Return
AL
  00H = Empty directory found
  FFH (255) = No empty directory available

DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for an empty entry or an existing entry for the specified filename.

If an empty directory entry is found, it is initialized to a zero-length file, the Open File system call (Function 0FH) is called, and AL returns 0. You can create a hidden file by using an extended FCB with the attribute byte (offset FCB-1) set to 2.

If an entry is found for the specified filename, all data in the file is released, making a zero-length file, and the Open File system call (Function 0FH) is issued for the filename (in other words, if you try to create a file that already exists, the existing file is erased, and a new, empty file is created).

If an empty directory entry is not found and there is no entry for the specified filename, AL returns FFH (255).

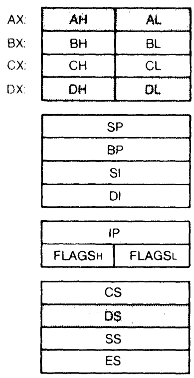
Macro Definition:
```
create  macro fcb
        mov   dx,offset fcb
        mov   ah,16H
        int   21H
        endm
```

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    14               ;offset of Record Size
                                        ;field of FCB
                .
                .
fcbl            db     2,"DIR     TMP"
                db     25  dup (?)
fcb2            db     2,"???????????"
                db     25  dup (?)
buffer          db     128 dup (?)
                .
                .
func_16H:       set_dta    buffer        ;see Function 1AH
                search_first  fcb2       ;see Function 11H
                cmp        al,FFH        ;directory entry found?
                je         all_done      ;no, no files on disk
                create     fcbl          ;THIS FUNCTION
                mov        fcbl[record_size],12
                                         ;set record size to 12
write it:       write_seq fcbl           ;see Function 15H
                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
```

Rename File (Function 17H)



Call
AH = 17H
DS:DX
  Modified FCB

Return
AL
  00H = Directory entry found
  FFH (255) = No directory entry found or destination already exists

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. The disk directory is searched for an entry that matches the first filename, which can contain the ? wild card character.

If a matching directory entry is found, the filename in the directory entry is changed to match the second filename in the modified FCB (the two filenames cannot be the same name). If the ? wild card character is used in the second filename, the corresponding characters in the filename of the directory entry are not changed. AL returns 0.

If a matching directory entry is not found or an entry is found for the second filename, AL returns FFH (255).

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

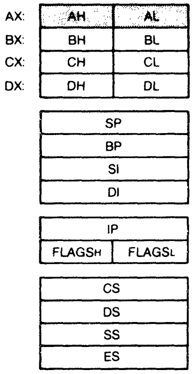
Example

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: $"
reply        db    17 dup(?)
crlf         db    13,10,"$"
                   .
                   .
```

```
func_17H:     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
```

Current Disk (Function 19H)



Call
AH = 19H

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

AL returns the currently selected drive (0 = A:, 1 = B:, etc.).

Macro Definition:

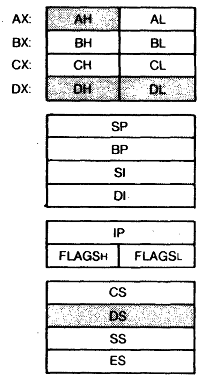
```
current_disk  macro
              mov   ah,19H
              int   21H
              endm
```

Example

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

```
message     db   "Current disk is $"  ;see Function 09H
                                      ;for explanation of $
crlf        db   13,10,"$"
            .
            .
func_19H:   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

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.

NOTE

> If you do not set the Disk Transfer Address, MS-DOS defaults to offset 80H in the Program Segment Prefix.

Macro Definition:

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

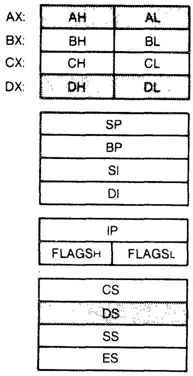
Example

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   14        ;offset of Record Size
                                  ;field of FCB
relative_record  equ    33        ;offset of Relative Record
                                  ;field of FCB
```

```
fcb             db    2,"ALPHABETDAT"
                db    25  dup (?)
buffer          db    34 dup(?),"$"
prompt          db   "Enter letter: $"
crlf            db    13,10,"$"
                .
                .
func_1AH:       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
```

Random Read (Function 21H)



Call
AH = 21H
DS:DX
  Opened FCB

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

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 loaded at the Disk Transfer Address.

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:

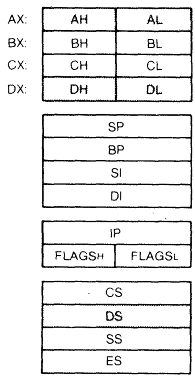
```
read_ran  macro   fcb
          mov     dx,offset fcb
          mov     ah,21H
          int     21H
          endm
```

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      equ   14        ;offset of Record Size
                                 ;field of FCB
relative_record  equ   33        ;offset of Relative Record
                                 ;field of FCB
                    .
                    .
fcb             db    2,"ALPHABETDAT"
                db    25  dup (?)
buffer          db    34 dup(?),"$"
prompt          db   "Enter letter: $"
crlf            db    13,10,"$"
                .
                .
func_21H:       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
  Opened FCB

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

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. If the record size is smaller than a sector (512 bytes), the records are buffered until a sector is ready to write.

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.

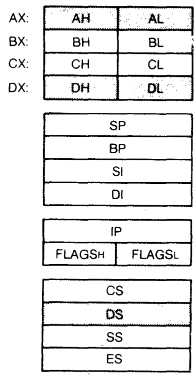
Macro Definition:
```
write_ran  macro  fcb
           mov    dx,offset fcb
           mov    ah,22H
           int    21H
           endm
```

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    14        ;offset of Record Size
                                 ;field of FCB
relative_record equ    33        ;offset of Relative Record
                                 ;field of FCB
           .
           .
fcb        db     2,"ALPHABETDAT"
           db     25  dup (?)
buffer     db     26 dup(?),13,10,"$"
prompt1    db    "Enter letter: $"
prompt2    db    "New record (RETURN for no change): $"
crlf       db     13,10,"$"
reply      db     28 dup (32)
blanks     db     26 dup (32)
           .
           .
func_22H:  set_dta  buffer              ;see Function 1AH
           open     fcb                 ;see Function 0FH
           mov      fcb[record_size],32 ;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
```

File Size (Function 23H)



Call
AH = 23H
DS:DX
  Unopened FCB

Return
AL
  00H = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (from the segment address in DS) of an unopened FCB. You must set the Record Size field (offset 0EH) to the proper value before calling this function. The disk directory is searched for the first matching entry.

If a matching directory entry is found, the Relative Record field (offset 21H) is set to the number of records in the file, calculated from the total file size in the directory entry (offset 1CH) and the Record Size field of the FCB (offset 0EH). AL returns 00.

If no matching directory is found, AL returns FFH (255).

NOTE

> If the value of the Record Size field of the FCB (offset 0EH) doesn't match the actual number of characters in a record, this function does not return the correct file size. If the default record size (128) is not correct, you must set the Record Size field to the correct value before using this function.

Macro Definition:

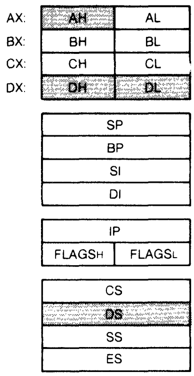
```
file_size macro  fcb
          mov    dx,offset fcb
          mov    ah,23H
          int    21H
          endm
```

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 file size and number of records in hexadecimal:

```
fcb            db       37 dup  (?)
prompt         db      "File name: $"
msg1           db      "Record length:     ",13,10,"$"
msg2           db      "Records:      ",13,10,"$"
crlf           db       13,10,"$"
reply          db       17 dup(?)
sixteen        db       16
               .
               .
func_23H:      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     si,33                ;offset to Relative
                                            ;Record field
               mov     di,9                 ;reply in msg_2
convert_it:    cmp     fcb[si],0            ;digit to convert?
               je      show_it              ;no, prepare message
               convert fcb[si],sixteen,msg_2[di]
               inc     si                   ;bump n-o-r index
               inc     di                   ;bump message index
               jmp     convert_it           ;check for a digit
show_it:       convert fcb[14],sixteen,msg_1[15]
               display msg_1                ;see Function 09H
               display msg_2                ;see Function 09H
               jmp     func_23H             ;get a filename
all_done:      close   fcb                  ;see Function 10H
```

Set Relative Record (Function 24H)



Call
AH = 24H
DS:DX
  Opened FCB

Return
None

DX must contain the offset (from the segment address in DS) of an opened FCB. The Relative Record field (offset 21H) is set to the same file address as the Current Block (offset 0CH) and Current Record (offset 20H) fields.

Macro Definition:

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

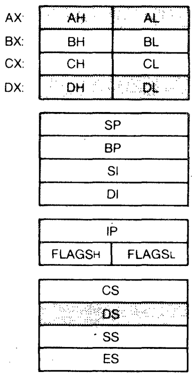
Example

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 (offset 0CH) and Current Record (offset 20H) fields:

```
current_record  equu   32           ;offset of Current Record
                                    ;field of FCB
file_size       equ    16           ;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    13,10,"$"
```

```
file_length  dw    ?
buffer    db       32767 dup(?)
          .
          .
func_24H: 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[file_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 Vector (Function 25H)



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

Return
None

Function 25H should be used to set a particular interrupt vector. The operating system can then manage the interrupts on a per-process basis. Note that programs should never set interrupt vectors by writing them directly in the low memory vector table.

DX must contain the offset (to the segment address in DS) of an interrupt-handling routine. AL must contain the number of the interrupt handled by the routine. The address in the vector table for the specified interrupt is set to DS:DX.

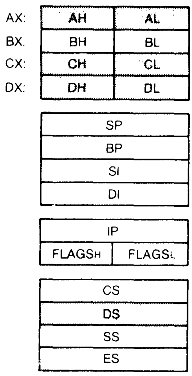
Macro Definition:

```
set_vector   macro   interrupt,seg_addr,off_addr
             mov     al,interrupt
             push    ds
             mov     ax,seg_addr
             mov     ds,ax
             mov     dx,off_addr
             mov     ah,25H
             int     21H
             pop     ds
             endm
```

Example

```
lds     dx,intvector
mov     ah,25H
mov     al,intnumber
int     21H
;There are no errors returned
```

Random Block Read (Function 27H)



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

Return
AL
  00H = Read completed successfully
  01H = EOF
  02H = End of segment
  03H = EOF, partial record
CX
  Number of blocks read

DX must contain the offset (to the segment address in DS) of an opened FCB. CX must contain the number of records to read; if it contains 0, the function returns without reading any records (no operation). The specified number of records -- calculated from the Record Size field (offset 0EH) -- is read starting at the record specified by the Relative Record field (offset 21H). The records are placed at the Disk Transfer Address.

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.

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

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
```

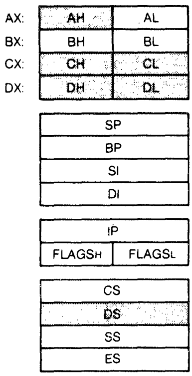
Example

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 length of 1 and a record count equal to the file size):

```
current_record  equ  32   ;offset of Current Record field
file_size       equ  16   ;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      13,10,"$"
file_length  dw   ?
buffer    db      32767 dup(?)
          .
          .
          .
func_27H: 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[file_size]
                                         ;get file size
          mov        file_length,ax      ;save it for
                                         ;ran_block_write
          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
```

```
mov           ax, file_length  ;get original file
                               ;size
ran_block_write  fcb,1,ax      ;see Function 28H
close         fcb              ;see Function 10H
```

Random Block Write (Function 28H)



Call
AH = 28H
DS:DX
  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

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. The specified number of records (calculated from the Record Size field, offset 0EH) is written from the Disk Transfer Address. The records are 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 the File Size field of the directory entry (offset 1CH) is set to the number of records specified by the Relative Record field of the FCB (offset 21H); allocation units are allocated or released, as required.

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 read one record; read canceled.

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

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
```

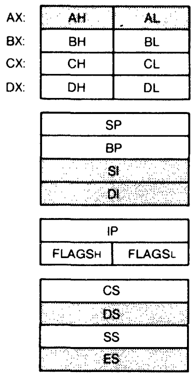
Example

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 count of 1 and a record length equal to file size):

```
current_record   equ  32     ;offset of Current Record field
file_size        equ  16     ;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      13,10,"$"
num_recs    dw      ?
buffer      db      32767 dup(?)
            .
            .
func_28H: 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[file_size]
                                       ;get file size
          mov         num_recs,ax      ;save it for
                                       ;ran_block_write
          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
mov         ax, file_length  ;get size of original
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
  String to parse
ES:DI
  Unopened FCB

Return
AL
  00H = No wild-card characters
  01H = Wild-card characters used
  FFH (255) = Drive letter invalid
DS:SI
  First byte past string that was parsed
ES:DI
  Unopened FCB

SI must contain the offset (to the segment address in DS) of a string (command line) to parse; DI must contain the offset (to the segment address in ES) of an unopened FCB. The string is parsed for a filename of the form d:filename.ext; if one is found, a corresponding unopened FCB is created at ES:DI.

Bits 0-3 of AL control the parsing and processing. Bits 4-7 are ignored:

Bit	Value	Meaning
0	0	All parsing stops if a file separator is encountered.
	1	Leading separators are ignored.
1	0	The drive number in the FCB is set to 0 (default drive) if the string does not contain a drive number.
	1	The drive number in the FCB is not changed if the string does not contain a drive number.
2	1	The filename in the FCB is not changed if the string does not contain a filename.
	0	The filename in the FCB is set to 8 blanks if the string does not contain a filename.
3	1	The extension in the FCB is not changed if the string does not contain an extension.
	0	The extension in the FCB is set to 3 blanks if the string does not contain an extension.

If the filename or extension 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 wild card character (* or ?); AL returns 0 if neither the filename nor extension contains a wild card character.

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

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

If the drive letter is invalid, AL returns FFH (255). If the string does not contain a valid filename, ES:DI+1 points to a blank (ASCII 32).

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, 1, 2, 3 on
      mov   ah,29H
      int   21H
      pop   es
      endm
```

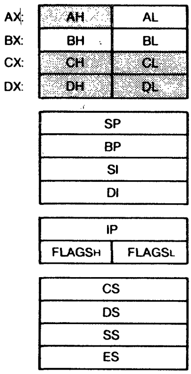
Example

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

```
fcb           db      37 dup (?)
prompt        db     "Filename: $"
reply         db      17 dup(?)
yes           db     "FILE EXISTS",13,10,"$"
no            db     "FILE DOES NOT EXIST",13,10,"$"
              .
              .
```

```
func_29H:      display     prompt          ;see Function 09H
               get_string  15,reply        ;see Function 0AH
               parse       reply[2],fcb    ;THIS FUNCTION
               search_first  fcb           ;see Function 11H
               cmp         al,FFH          ;dir. entry found?
               je          not_there       ;no
               display     yes             ;see Function 09H
               jmp         continue
not_there:     display     no
continue:      .
               .
```

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.)

This function 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
```

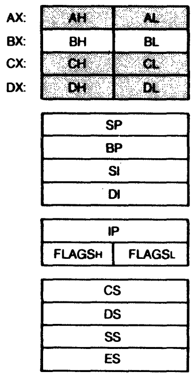
Example

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
            .
            .
func_2AH:   get_date              ;see above
            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
```

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 (255) = Date was invalid

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 (255).

Macro Definition:

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

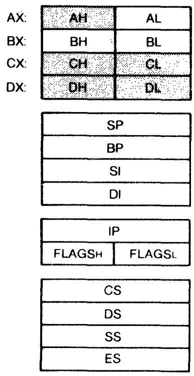
Example

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
            .
            .
func_2BH:   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)

This function 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)

Macro Definition:

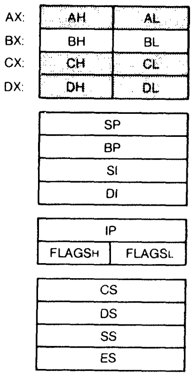
```
get_time  macro
          mov   ah,2CH
          int   21H
          endm
```

Example

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

```
time          db    "00:00:00.00",13,10,"$"
ten           db     10
              .
              .
func_2CH:     get_time                  ;THIS FUNCTION
              convert ch,ten,time       ;see end of chapter
              convert cl,ten,time[3]    ;see end of chapter
              convert dh,ten,time[6]    ;see end of chapter
              convert dl,ten,time[9]    ;see end of chapter
              display time              ;see Function 09H
              check_kbd_status          ;see Function 0BH
              cmp      al,FFH           ;has a key been pressed?
              je       all_done         ;yes, terminate
              jmp      func_2CH         ;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 (255) = Time was invalid

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)

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 (255).

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
```

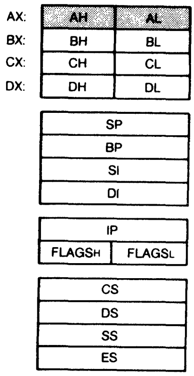
Example

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",13,10,"$"
ten           db   10
              .
              .
func_2DH:     set_time  0,0,0,0       ;THIS FUNCTION
read_clock:   get_time                ;see Function 2CH
```

```
              convert  ch,ten,time     ;see end of chapter
              convert  cl,ten,time[3]  ;see end of chapter
              convert  dh,ten,time[6]  ;see end of chapter
              convert  dl,ten,time[9]  ;see end of chapter
              display  time            ;see Function 09H
              dir_console_io  FFH      ;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      func_2DH        ;keep displaying time
```

Set/Reset Verify Flag (Function 2EH)



Call
AH = 2EH
AL
  00H = Do not verify
  01H = Verify

Return
None

AL must be either 1 (verify after each disk write) or 0 (write without verifying). 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.

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

Example

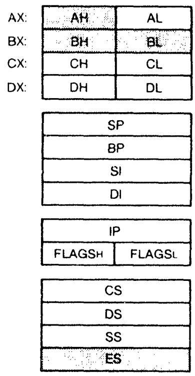
The following program copies the contents of a single-sided disk in drive A: to the disk in drive B:, verifying each write. It uses a buffer of 32K bytes:

```
on          equ   1
off         equ   0
            .
            .
prompt      db    "Source in A, target in B",13,10
            db    "Any key to start. $"
start       dw    0
buffer      db    64 dup (512 dup(?))    ;64 sectors
            .
            .
func_2DH:   display prompt               ;see Function 09H
            read_kbd                     ;see Function 08H
            verify on                    ;THIS FUNCTION
```

```
             mov     cx,5                     ;copy 64 sectors
                                              ;5 times
copy:        push    cx                       ;save counter
             abs_disk_read   0,buffer,64,start
                                              ;see Interrupt 25H
             abs_disk_write  1,buffer,64,start
                                              ;see Interrupt 26H
             add     start,64                 ;do next 64 sectors
             pop     cx                       ;restore counter
             loop    copy                     ;do it again
             verify off                       ;THIS FUNCTION

disk_ read  0,buffer,64,start                 ;see Interrupt 25H
             abs_disk_write  1,buffer,64,start
                                              ;see Interrupt 26H
             add     start,64                 ;do next 64 sectors
             pop     cx                       ;restore counter
             loop    copy                     ;do it again
             verify off
```

Get Disk Transfer Address (Function 2FH)



Call
AH = 2FH

Return
ES:BX
  Points to Disk Transfer Address

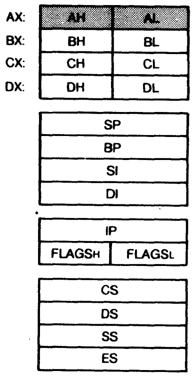
Function 2FH returns the DMA transfer address.

Error returns:
  None.

Example

```
mov     ah,2FH
int     21H
        ;es:bx has current DMA transfer address
```

Get DOS Version Number (Function 30H)



Call
AH = 30H

Return
AL
  Major version number
AH
  Minor version number

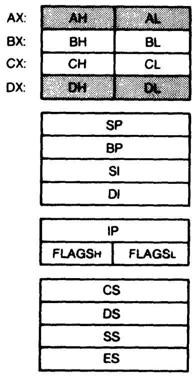
This function returns the MS-DOS version number. On return, AL.AH will be the two-part version designation; i.e., for MS-DOS 1.28, AL would be 1 and AH would be 28. For pre-1.28, DOS AL = 0. Note that version 1.1 is the same as 1.10, not the same as 1.01.

Error returns:
  None.

Example

```
    mov     ah,30H
    int     21H
          ; al is the major version number
          ; ah is the minor version number
          ; bh is the OEM number
          ; bl:cx is the (24 bit) user number
```

Keep Process (Function 31H)



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

Return
None

This call terminates the current process and attempts to set the initial allocation block to a specific size in paragraphs. It will not free up any other allocation blocks belonging to that process. The exit code passed in AX is retrievable by the parent via Function 4DH.

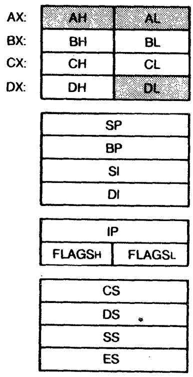
This method is preferred over Interrupt 27H and has the advantage of allowing more than 64K to be kept.

    Error returns:
        None.

Example

```
mov     al, exitcode
mov     dx, parasize
mov     ah, 31H
int     21H
```

CONTROL-C Check (Function 33H)



Call
AH = 33H
AL
  Function
  00H = Request current state
  01H = Set state
DL (if setting)
  00H = Off
  01H = On

Return
DL
  00H = Off
  01H = On

MS-DOS ordinarily checks for a CONTROL-C on the controlling device only when doing function call operations 01H-0CH to that device. Function 33H allows the user to expand this checking to include any system call. For example, with the CONTROL-C trapping off, all disk I/O will proceed without interruption; with CONTROL-C trapping on, the CONTROL-C interrupt is given at the system call that initiates the disk operation.

NOTE

> Programs that wish to use calls 06H or 07H to read CONTROL-Cs as data must ensure that the CONTROL-C check is off.

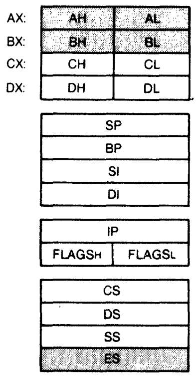
Error return:
AL = FF
  The function passed in AL was not in the range 0:1.

Example

```
mov     dl,val
mov     ah,33H
mov     al,func
```

```
        int     21H
                ; If al was 0, then dl has the current value
                ;of the CONTROL-C check
```

Get Interrupt Vector (Function 35H)



Call
AH = 35H
AL
  Interrupt number

Return
ES:BX
  Pointer to interrupt routine

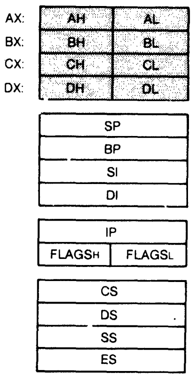
This function returns the interrupt vector associated with an interrupt. Note that programs should never get an interrupt vector by reading the low memory vector table directly.

Error returns:
  None.

Example

```
mov     ah,35H
mov     al,interrupt
int     21H
   ; es:bx now has long pointer to interrupt routine
```

Get Disk Free Space (Function 36H)



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

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

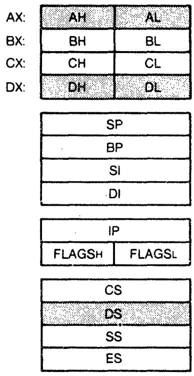
This function returns free space on disk along with additional information about the disk.

Error returns:
AX = FFFF
  The drive number given in DL was invalid.

Example

```
mov     ah,36H
mov     dl,Drive            ;0 = default, A = 1
int     21H
   ; bx = Number of free allocation units on drive
   ; dx = Total number of allocation units on drive
   ; cx = Bytes per sector
   ; ax = Sectors per allocation unit
```

Return Country-Dependent Information (Function 38H)



Call
AH = 38H
DS:DX
  Pointer to 32-byte memory area
AL
  Function code. In MS-DOS 2.0,
  must be 0

Return
Carry set:
AX
  2 = file not found
Carry not set:
  DX:DS filled in with country data

The value passed in AL is either 0 (for current country) or a country code. Country codes are typically the international telephone prefix code for the country.

If DX = -1, then the call sets the current country (as returned by the AL=0 call) to the country code in AL. If the country code is not found, the current country is not changed.

NOTE

Applications must assume 32 bytes of information. This means the buffer pointed to by DS:DX must be able to accommodate 32 bytes.

This function is fully supported only in versions of MS-DOS 2.01 and higher. It exists in MS-DOS 2.0, but is not fully implemented.

This function returns, in the block of memory pointed to by DS:DX, the following information pertinent to international applications:

WORD Date/time format
5 BYTE ASCIZ string currency symbol
2 BYTE ASCIZ string thousands separator
2 BYTE ASCIZ string decimal separator
2 BYTE ASCIZ string date separator
2 BYTE ASCIZ string time separator
1 BYTE Bit field
1 BYTE Currency places
1 BYTE time format
DWORD Case Mapping call
2 BYTE ASCIZ string data list separator

The format of most of these entries is ASCIZ (a NUL terminated ASCII string), but a fixed size is allocated for each field for easy indexing into the table.

The date/time format has the following values:

```
0 - USA standard    h:m:s m/d/y
1 - Europe standard h:m:s d/m/y
2 - Japan standard  y/m/d h:m:s
```

The bit field contains 8 bit values. Any bit not currently defined must be assumed to have a random value.

```
Bit 0 = 0 If currency symbol precedes the
          currency amount.
      = 1 If currency symbol comes after
          the currency amount.
Bit 1 = 0 If the currency symbol immediately
          precedes the currency amount.
      = 1 If there is a space between the
          currency symbol and the amount.
```

The time format has the following values:

    0 - 12 hour time
    1 - 24 hour time

The currency places field indicates the number of places which appear after the decimal point on currency amounts.

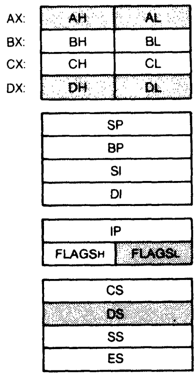
The Case Mapping call is a FAR procedure which will perform country specific lower-to-uppercase mapping on character values from 80H to FFH. It is called with the character to be mapped in AL. It returns the correct upper case code for that character, if any, in AL. AL and the FLAGS are the only registers altered. It is allowable to pass this routine code below 80H; however nothing is done to characters in this range. In the case where there is no mapping, AL is not altered.

```
Error returns:
AX
 2  = file not found
    The country passed in AL was not found (no
    table for specified country).
```

Example

```
 lds     dx, blk
 mov     ah, 38H
 mov     al, Country_code
 int     21H
    ;AX = Country code of country returned
```

Create Sub-Directory (Function 39H)



Call
AH = 39H
DX:DS
  Pointer to pathname

Return
Carry set:
AX
  3 = path not found
  5 = access denied
Carry not set:
  No error

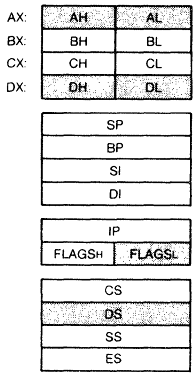
Given a pointer to an ASCIZ name, this function creates a new directory entry at the end.

Error returns:
AX
3 = path not found
    The path specified was invalid or not found.
5 = access denied
    The directory could not be created (no room in parent directory), the directory/file already existed or a device name was specified.

Example

```
lds     dx, name
mov     ah, 39H
int     21H
```

Remove a Directory Entry (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

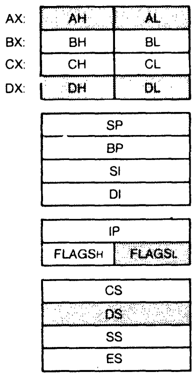
Function 3AH is given an ASCIZ name of a directory. That directory is removed from its parent directory.

Error returns:
AX
3 = path not found
  The path specified was invalid or not found.
5 = access denied
  The path specified was not empty, not a directory, the root directory, or contained invalid information.
16 = current directory
  The path specified was the current directory on a drive.

Example

```
lds     dx, name
mov     ah, 3AH
int     21H
```

Change the 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

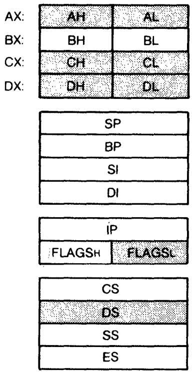
Function 3BH is given the ASCIZ name of the directory which is to become the current directory. If any member of the specified pathname does not exist, then the current directory is unchanged. Otherwise, the current directory is set to the string.

```
    Error returns:
    AX
     3 = path not found
            The path specified in DS:DX either indicated a
            file or the path was invalid.
```

Example

```
lds     dx, name
mov     ah, 3BH
int     21H
```

Create a File (Function 3CH)



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

Return
Carry set:
AX
  5 = access denied
  3 = path not found
  4 = too many open files
Carry not set:
  AX is handle number

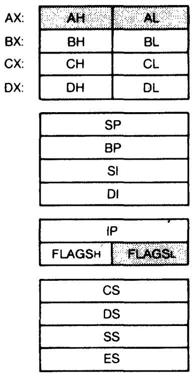
Function 3CH creates a new file or truncates an old file to zero length in preparation for writing. If the file did not exist, then the file is created in the appropriate directory and the file is given the attribute found in CX. The file handle returned has been opened for read/write access.

Error returns:
AX
 5 = access denied
  The attributes specified in CX contained one that could not be created (directory, volume ID), a file already existed with a more inclusive set of attributes, or a directory existed with the same name.
 3 = path not found
  The path specified was invalid.
 4 = too many open files
  The file was created with the specified attributes, but there were no free handles available for the process, or the internal system tables were full.

Example

```
lds     dx, name
mov     ah, 3CH
mov     cx, attribute
int     21H
   ; ax now has the handle
```

Open a File (Function 3DH)



Call
AH = 3DH
AL
  Access
  0 = File opened for reading
  1 = File opened for writing
  2 = File opened for both
  reading and writing

Return
Carry set:
AX
  12 = invalid access
  2 = file not found
  5 = access denied
  4 = too many open files
Carry not set:
  AX is handle number

Function 3DH associates a 16-bit file handle with a file.

The following values are allowed:

ACCESS	Function
0	file is opened for reading
1	file is opened for writing
2	file is opened for both reading and writing.

DS:DX point to an ASCIZ name of the file to be opened.

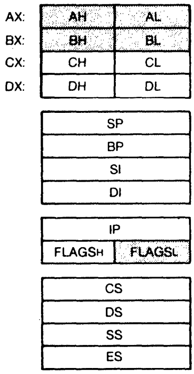
The read/write pointer is set at the first byte of the file and the record size of the file is 1 byte. The returned file handle must be used for subsequent I/O to the file.

Error returns:
AX
12 = invalid access
    The access specified in AL was not in the range 0:2.
2 = file not found
    The path specified was invalid or not found.
5 = access denied
    The user attempted to open a directory or volume-id, or open a read-only file for writing.
4 = too many open files
    There were no free handles available in the current process or the internal system tables were full.

Example

```
        lds     dx, name
        mov     ah, 3DH
        mov     al, access
        int     21H
            ; ax has error or file handle
            ; If successful open
```

Close a File Handle (Function 3EH)



Call
AH = 3EH
BX
  File handle

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

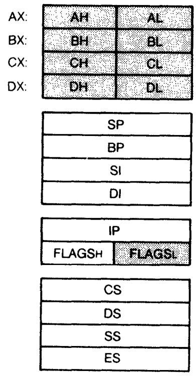
In BX is passed a file handle (like that returned by Functions 3DH, 3CH, or 45H), Function 3EH closes the associated file. Internal buffers are flushed.

Error return:
AX
6 = invalid handle
The handle passed in BX was not currently open.

Example

```
mov     bx, handle
mov     ah, 3EH
int     21H
```

Read From File/Device (Function 3FH)



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

Return
Carry set:
AX
  Number of bytes read
  6 = invalid handle
  5 = error set:
Carry not set:
  AX = number of bytes read

Function 3FH transfers count bytes from a file into a buffer location. It is not guaranteed that all count bytes will be read; for example, reading from the keyboard will read at most one line of text. If the returned value is zero, then the program has tried to read from the end of file.

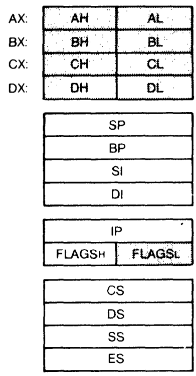
All I/O is done using normalized pointers; no segment wraparound will occur.

Error returns:
AX
 6 = invalid handle
    The handle passed in BX was not currently open.
 5 = access denied
    The handle passed in BX was opened in a mode that did not allow reading.

Example

```
lds     dx, buf
mov     cx, count
mov     bx, handle
mov     ah, 3FH
int     21H
   ; ax has number of bytes read
```

Write to a File or Device (Function 40H)



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

Return
Carry set:
AX
  Number of bytes written
  6 = invalid handle
  5 = access denied
Carry not set:
AX = number of bytes written

Function 40H transfers count bytes from a buffer into a file. It should be regarded as an error if the number of bytes written is not the same as the number requested.

The write system call with a count of zero (CX = 0) will set the file size to the current position. Allocation units are allocated or released as required.

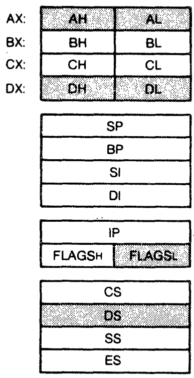
All I/O is done using normalized pointers; no segment wraparound will occur.

Error returns:
AX
 6 = invalid handle
  The handle passed in BX was not currently open.
 5 = access denied
  The handle was not opened in a mode that allowed writing.

Example

```
lds     dx, buf
mov     cx, count
mov     bx, handle
mov     ah, 40H
int     21H
   ;ax has number of bytes written
```

Delete a Directory Entry (Function 41H)



Call
AH = 41H
DS:DX
  Pointer to path name

Return
Carry set:
AX
  2 = file not found
  5 = access denied
Carry not set:
  No error

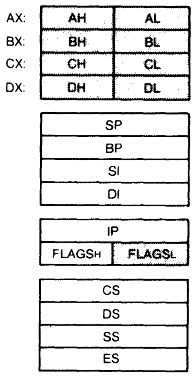
Function 41H removes a directory entry associated with a filename.

Error returns:
AX
2 = file not found
  The path specified was invalid or not found.
5 = access denied
  The path specified was a directory or read-only.

Example

```
lds     dx, name
mov     ah, 41H
int     21H
```

Move File Pointer (Function 42H)



Call
AH = 42H
CX:DX
  Distance to move, in bytes
AL
  Method of moving:
  (see text)
BX
  File handle

Return
Carry set:
AX
  6 = invalid handle
  1 = invalid function
Carry not set:
  DX:AX = new pointer location

Function 42H moves the read/write pointer according to one of the following methods:

Method	Function
0	The pointer is moved to offset bytes from the beginning of the file.
1	The pointer is moved to the current location plus offset.
2	The pointer is moved to the end of file plus offset.

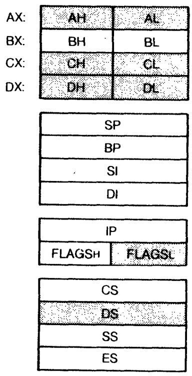
Offset should be regarded as a 32-bit integer with CX occupying the most significant 16 bits.

Error returns:
AX
 6 = invalid handle
   The handle passed in BX was not currently open.
 1 = invalid function
   The function passed in AL was not in the range 0:2.

Example

```
mov     dx, offsetlow
mov     cx, offsethigh
mov     al, method
mov     bx, handle
mov     ah, 42H
int     21H
   ; dx:ax has the new location of the pointer
```

Change Attributes (Function 43H)



Call
AH = 43H
DS:DX
  Pointer to path name
CX (if AL = 01)
  Attribute to be set
AL
  Function
  01 Set to CX
  00 Return in CX

Return
Carry set:
AX
  3 = path not found
  5 = access denied
  1 = invalid function
Carry not set:
  CX attributes (if AL = 00)

Given an ASCIZ name, Function 42H will set/get the attributes of the file to those given in CX.

A function code is passed in AL:

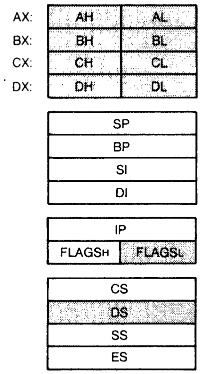
```
AL  Function
--  --------
0  Return the attributes of the file in CX.
1  Set the attributes of the file to those in CX.
```

Error returns:
AX
 3 = path not found
  The path specified was invalid.
 5 = access denied
  The attributes specified in CX contained one that could not be changed (directory, volume ID).
 1 = invalid function
  The function passed in AL was not in the range 0:1.

Example

```
lds     dx, name
mov     cx, attribute
mov     al, func
int     ah, 43H
int     21H
```

I/O Control for Devices (Function 44H)



Call
AH = 44H
BX
  Handle
BL
  Drive (for calls AL = 4, 5
  0 = default, 1 = A, etc.)
DS:DX
  Data or buffer
CX
  Bytes to read or write
AL
  Function code; see text

Return
Carry set:
AX
  6 = invalid handle
  1 = invalid function
  13 = invalid data
  5 = access denied
Carry not set:
AL = 2,3,4,5
AX = Count transferred
AL = 6,7
    00 = Not ready
    FF = Ready

Function 44H sets or gets device information associated with an open handle, or sends/receives a control string to a device handle or device.

The following values are allowed for function:

Request	Function
0	Get device information (returned in DX)
1	Set device information (as determined by DX)
2	Read CX number of bytes into DS:DX from device control channel
3	Write CX number of bytes from DS:DX to device control channel
4	Same as 2 only drive number in BL 0=default,A:=1,B:=2,...
5	Same as 3 only drive number in BL 0=default,A:=1,B:=2,...
6	Get input status
7	Get output status

This function can be used to get information about device channels. Calls can be made on regular files, but only calls 0,6 and 7 are defined in that case (AL=0,6,7). All other calls return an invalid function error.

Calls AL=0 and AL=1

The bits of DX are defined as follows for calls AL=0 and AL=1. Note that the upper byte MUST be zero on a set call.

15	14	13 12 11 10 9 8	7	6	5	4	3	2	1	0
Res	CTRL	Reserved	ISDEV	EOF	RAW	SPECL	ISCLK	ISNUL	ISCOT	ISCIN

ISDEV = 1 if this channel is a device
= 0 if this channel is a disk file (Bits 8-15 = 0 in this case)

If ISDEV = 1

```
EOF   = 0 if End Of File on input
RAW   = 1 if this device is in Raw mode
      = 0 if this device is cooked
ISCLK = 1 if this device is the clock device
ISNUL = 1 if this device is the null device
ISCOT = 1 if this device is the console output
ISCIN = 1 if this device is the console input
SPECL = 1 if this device is special
```

CTRL = 0 if this device can not do control strings via calls AL=2 and AL=3.
CTRL = 1 if this device can process control strings via calls AL=2 and AL=3.
NOTE that this bit cannot be set.

If ISDEV = 0
EOF = 0 if channel has been written
Bits 0-5 are the block device number for the channel (0 = A:, 1 = B:, ...)

Bits 15,8-13,4 are reserved and should not be altered.

Calls 2..5:

These four calls allow arbitrary control strings to be sent or received from a device. The call syntax is the same as the read and write calls, except for 4 and 5, which take a drive number in BL instead of a handle in BX.

An invalid function error is returned if the CTRL bit (see above) is 0.

An access denied is returned by calls AL=4,5 if the drive number is invalid.

Calls 6,7:

These two calls allow the user to check if a file handle is ready for input or output. Status of handles open to a device is the intended use of these calls, but status of a handle open to a disk file is allowed, and is defined as follows:

Input:
Always ready (AL=FF) until EOF reached, then always not ready (AL=0) unless current position changed via LSEEK.

Output:
Always ready (even if disk full).

IMPORTANT

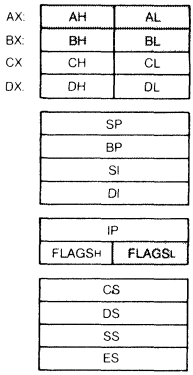
The status is defined at the time the system is CALLED. On future versions, by the time control is returned to the user from the system, the status returned may NOT correctly reflect the true current state of the device or file.

Error returns:
AX
6 = invalid handle
The handle passed in BX was not currently open.
1 = invalid function
The function passed in AL was not in the range 0:7.
13 = invalid data
5 = access denied (calls AL=4..7)

Example

```
    mov     bx, Handle
 (or mov    bl, drive    for calls AL=4,5
                         0=default,A:=1...)
    mov     dx, Data
 (or lds    dx, buf      and
    mov     cx, count    for calls AL=2,3,4,5)
    mov     ah, 44H
    mov     al, func
    int     21H
   ; For calls AL=2,3,4,5 AX is the number of bytes
   ; transferred (same as READ and WRITE).
   ; For calls AL=6,7 AL is status returned, AL=0 if
   ; status is not ready, AL=0FFH otherwise.
```

Duplicate a File Handle (Function 45H)



Call
AH = 45H
BX
  File handle

Return
Carry set:
AX
  6 = invalid handle
  4 = too many open files
Carry not set:
AX = new file handle

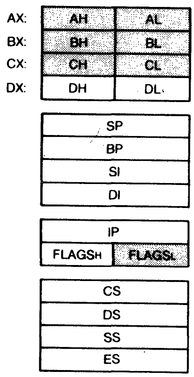
Function 45H takes an already opened file handle and returns a new handle that refers to the same file at the same position.

Error returns:
AX
6 = invalid handle
    The handle passed in BX was not currently open.
4 = too many open files
    There were no free handles available in the current process or the internal system tables were full.

Example

```
mov     bx, fh
mov     ah, 45H
int     21H
    ; ax has the returned handle
```

Force a Duplicate of a Handle (Function 46H)



Call
AH = 46H
BX
  Existing file handle
CX
  New file handle

Return
Carry set:
AX
  6 = invalid handle
  4 = too many open files
Carry not set:
  No error

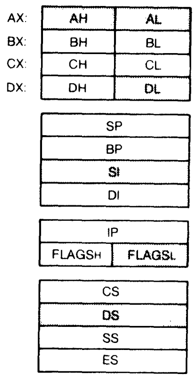
Function 46H takes an already opened file handle and returns a new handle that refers to the same file at the same position. If there was already a file open on handle CX, it is closed first.

Error returns:
AX
 6 = invalid handle
   The handle passed in BX was not currently open.
 4 = too many open files
   There were no free handles available in the current process or the internal system tables were full.

Example

```
mov     bx, fh
mov     cx, newfh
mov     ah, 46H
int     21H
```

Return Text of 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
Carry not set:
  No error

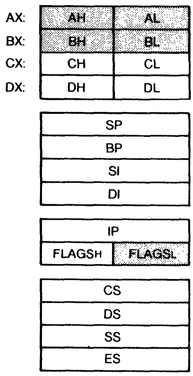
Function 47H returns the current directory for a particular drive. The directory is root-relative and does not contain the drive specifier or leading path separator. The drive code passed in DL is 0=default, 1=A:, 2=B:, etc.

Error returns:
AX
15 = invalid drive
    The drive specified in DL was invalid.

Example

```
mov     ah, 47H
lds     si,area
mov     dl,drive
int     21H
    ; ds:si is a pointer to 64 byte area that
    ; contains drive current directory.
```

Allocate Memory (Function 48H)



Call
AH = 48H
BX
  Size of memory to be allocated

Return
Carry set:
AX
  8 = not enough memory
  7 = arena trashed
BX
  Maximum size that could be allocated
Carry not set:
AX:0
  Pointer to the allocated memory

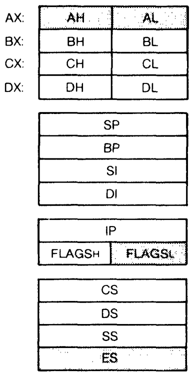
Function 48H returns a pointer to a free block of memory that has the requested size in paragraphs.

Error return:
AX
8 = not enough memory
  The largest available free block is smaller than that requested or there is no free block.
7 = arena trashed
  The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it.

Example

```
mov     bx,size
mov     ah,48H
int     21H
   ; ax:0 is pointer to allocated memory
   ; if alloc fails, bx is the largest block available
```

Free Allocated Memory (Function 49H)



Call
AH = 49H
ES
  Segment address of memory
  area to be freed

Return
Carry set:
AX
  9 = invalid block
  7 = arena trashed
Carry not set:
  No error

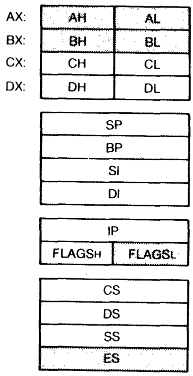
Function 49H returns a piece of memory to the system pool that was allocated by Function Request 49H.

Error return:
AX
 9 = invalid block
   The block passed in ES is not one allocated via Function Request 49H.
 7 = arena trashed
   The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it.

Example

```
mov     es,block
mov     ah,49H
int     21H
```

Modify Allocated Memory Blocks (Function 4AH)



Call
AH = 4AH
ES
  Segment address of memory area
BX
  Requested memory area size

Return
Carry set:
AX
  9 = invalid block
  7 = arena trashed
  8 = not enough memory
BX
  Maximum size possible
Carry not set:
  No error

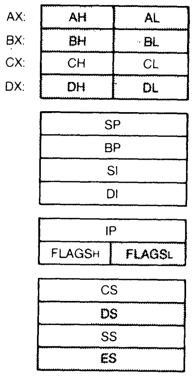
Function 4AH will attempt to grow/shrink an allocated block of memory.

Error return:
AX
 9 = invalid block
  The block passed in ES is not one allocated via this function.
 7 = arena trashed
  The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it.
 8 = not enough memory
  There was not enough free memory after the specified block to satisfy the grow request.

Example

```
        mov     es,block
        mov     bx,newsize
        mov     ah,4AH
        int     21H
           ; if setblock  fails  for  growing, BX will have the
           ; maximum size possible
```

Load and Execute a Program (Function 4BH)



Call
AH = 4BH
DS:DX
  Pointer to pathname
ES:BX
  Pointer to parameter block
AL
  00 = Load and execute program
  03 = Load program

Return
Carry set:
AX
  1 = invalid function
  10 = bad environment
  11 = bad format
  8 = not enough memory
  2 = file not found
Carry not set:
  No error

This function allows a program to load another program into memory and (default) begin execution of it. DS:DX points to the ASCIZ name of the file to be loaded. ES:BX points to a parameter block for the load.

A function code is passed in AL:

AL	Function
0	Load and execute the program. A program header is established for the program and the terminate and CONTROL-C addresses are set to the instruction after the EXEC system call.
3	Load (do not create) the program header, and do not begin execution. This is useful in loading program overlays.

For each value of AL, the block has the following format:

AL = 0 -> load/execute program

WORD segment address of environment.
DWORD pointer to command line at 80H
DWORD pointer to default FCB to be passed at 5CH
DWORD pointer to default FCB to be passed at 6CH

AL = 3 -> load overlay

WORD segment address where file will be loaded.
WORD relocation factor to be applied to the image.

Note that all open files of a process are duplicated in the child process after an EXEC. This is extremely powerful; the parent process has control over the meanings of stdin, stdout, stderr, stdaux and stdprn. The parent could, for example, write a series of records to a file, open the file as standard input, open a listing file as standard output and then EXEC a sort program that takes its input from stdin and writes to stdout.

Also inherited (or passed from the parent) is an "environment." This is a block of text strings (less than 32K bytes total) that convey various configuration parameters. The format of the environment is as follows:

(paragraph boundary)

BYTE ASCIZ string 1
BYTE ASCIZ string 2
...
BYTE ASCIZ string n
BYTE of zero

Typically the environment strings have the form:

    parameter=value

For example, COMMAND.COM might pass its execution search path as:

    PATH=A:\BIN;B:\BASIC\LIB

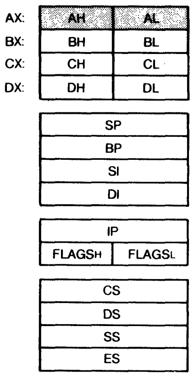
A zero value of the environment address causes the child process to inherit the parent's environment unchanged.

Error returns:
AX
1 = invalid function
    The function passed in AL was not 0, 1 or 3.
10 = bad environment
    The environment was larger than 32Kb.
11 = bad format
    The file pointed to by DS:DX was an EXE format file and contained information that was internally inconsistent.
8 = not enough memory
    There was not enough memory for the process to be created.
2 = file not found
    The path specified was invalid or not found.

Example

```
lds     dx, name
les     bx, blk
mov     ah, 4BH
mov     al, func
int     21H
```

Terminate a Process (Function 4CH)



Call
AH = 4CH
AL
  Return code

Return
None

Function 4CH terminates the current process and transfers control to the invoking process. In addition, a return code may be sent. All files open at the time are closed.

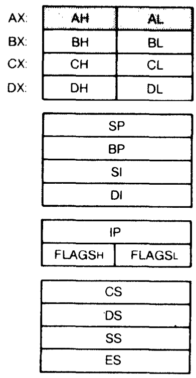
This method is preferred over all others (Interrupt 20H, JMP 0) and has the advantage that CS:0 does not have to point to the Program Header Prefix.

Error returns:
    None.

Example

```
mov     al, code
mov     ah, 4CH
int     21H
```

Retrieve the Return Code of a Child (Function 4DH)



Call
AH = 4DH

Return
AX
Exit code

Function 4DH returns the Exit code specified by a child process. It returns this Exit code only once. The low byte of this code is that sent by the Exit routine. The high byte is one of the following:

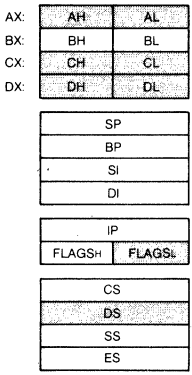
0 - Terminate/abort
1 - CONTROL-C
2 - Hard error
3 - Terminate and stay resident

Error returns:
None.

Example

```
mov     ah, 4DH
int     21H
    ; ax has the exit code
```

Find Match File (Function 4EH)



Call
AH = 4EH
DS:DX
  Pointer to pathname
CX
  Search attributes

Return
Carry set:
AX
  2 = file not found
  18 = no more files
Carry not set:
  No error

Function 4EH takes a pathname with wild-card characters in the last component (passed in DS:DX), a set of attributes (passed in CX) and attempts to find all files that match the pathname and have a subset of the required attributes. A datablock at the current DMA is written that contains information in the following form:

```
find_buf_reserved   DB  21 DUP (?); Reserved*
find_buf_attr       DB  ?  ; attribute found
find_buf_time       DW  ?  ; time
find_buf_date       DW  ?  ; date
find_buf_size_l     DW  ?  ; low(size)
find_buf_size_h     DW  ?  ; high(size)
find_buf_pname      DB  13 DUP (?) ; packed name
find_buf      ENDS
```

*Reserved for MS-DOS use on subsequent find_nexts

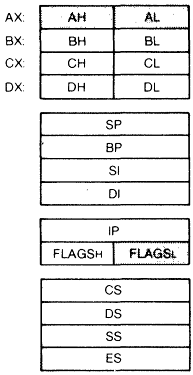
To obtain the subsequent matches of the pathname, see the description of Function 4FH.

Error returns:
AX
2 = file not found
  The path specified in DS:DX was an invalid path.
18 = no more files
  There were no files matching this specification.

Example

```
mov ah, 4EH
lds dx, pathname
mov cx, attr
int 21H
   ; dma address has datablock
```

Step Through a Directory Matching Files (Function 4FH)



Call
AH = 4FH

Return
Carry set:
AX
  18 = no more files
Carry not set:
  No error

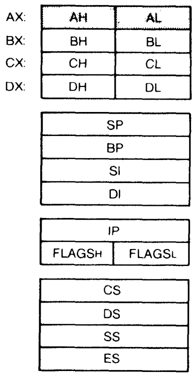
Function 4FH finds the next matching entry in a directory. The current DMA address must point at a block returned by Function 4EH (see Function 4EH).

Error returns:
AX
18 = no more files
    There are no more files matching this pattern.

Example

```
    ; dma points at area returned by Function 4FH
mov ah, 4FH
int 21H
    ; next entry is at dma
```

Return Current Setting of Verify After Write Flag (Function 54H)



Call
AH = 54H

Return
AL
  Current verify flag value

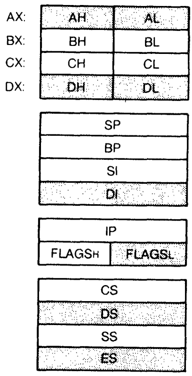
The current value of the verify flag is returned in AL.

Error returns:
  None.

Example

```
mov     ah,54H
int     21H
   ; al is the current verify flag value
```

Move a Directory Entry (Function 56H)



Call
AH = 56H
DS:DX
  Pointer to pathname of existing file
ES:DI
  Pointer to new pathname

Return
Carry set:
AX
  2 = file not found
  17 = not same device
  5 = access denied
Carry not set:
  No error

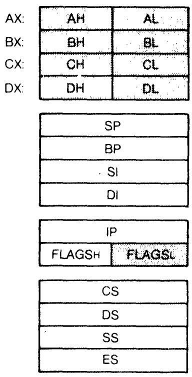
Function 56H attempts to rename a file into another path. The paths must be on the same device.

Error returns:
AX
 2 = file not found
  The file name specifed by DS:DX was not found.
17 = not same device
  The source and destination are on different drives.
 5 = access denied
  The path specified in DS:DX was a directory or the file specified by ES:DI exists or the destination directory entry could not be created.

Example

```
lds     dx, source
les     di, dest
mov     ah, 56H
int     21H
```

Get/Set Date/Time of File (Function 57H)



Call
AH = 57H
AL
  00 = get date and time
  01 = set date and time
BX
  File handle
CX (if AL = 01)
  Time to be set
DX (if AL = 01)
  Date to be set

Return
Carry set:
AX
  1 = invalid function
  6 = invalid handle
Carry not set:
No error
CX/DX set if function 0

Function 57H returns or sets the last-write time for a handle. These times are not recorded until the file is closed.

A function code is passed in AL:

```
AL  Function
--  --------
 0  Return the time/date of the handle in CX/DX
 1  Set the time/date of the handle to CX/DX
```

Error returns:
AX
 1 = invalid function
   The function passed in AL was not in the range 0:1.
 6 = invalid handle
   The handle passed in BX was not currently open.

Example

```
mov ah, 57H
mov al, func
mov bx, handle
   ; if al = 1 then then next two are mandatory
mov cx, time
mov dx, date
int 21H
   ; if al = 0 then cx/dx has the last write time/date
   ; for the handle.
```

1.8 MACRO DEFINITIONS FOR MS-DOS SYSTEM CALL EXAMPLES

NOTE

These macro definitions apply to system call examples 00H through 57H.

```
.xlist
;
;*******************
; Interrupts
;*******************
;
                                        ;ABS_DISK_READ
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       37                  ;interrupt 37
          popf
          endm
;
                                        ;ABS_DISK_WRITE
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       38                  ;interrupt 38
          popf
          endm
;
stay_resident  macro  last_instruc      ;STAY_RESIDENT
          mov       dx,offset last_instruc
          inc       dx
          int       39                  ;interrupt 39
          endm
;
;*******************
; Functions
;*******************
;
read_kbd_and_echo  macro                ;READ_KBD_AND_ECHO
          mov       ah,1                ;function 1
          int       33
          endm
;
display_char  macro  character          ;DISPLAY_CHAR
          mov       dl,character
```

```
          mov       ah,2                ;function 2
          int       33
          endm
;
aux_input macro                         ;AUX_INPUT
          mov       ah,3                ;function 3
          int       33
          endm
;
aux_output macro                        ;AUX_OUTPUT
          mov       ah,4                ;function 4
          int       33
          endm
;;page
print_char  macro   character           ;PRINT_CHAR
          mov       dl,character
          mov       ah,5                ;function 5
          int       33
          endm
;
dir_console_io macro switch             ;DIR_CONSOLE_IO
          mov       dl,switch
          mov       ah,6                ;function 6
          int       33
          endm
;
dir_console_input macro                 ;DIR_CONSOLE_INPUT
          mov       ah,7                ;function 7
          int       33
          endm
;
read_kbd  macro                         ;READ_KBD
          mov       ah,8                ;function 8
          int       33
          endm
;
display   macro     string              ;DISPLAY
          mov       dx,offset string
          mov       ah,9                ;function 9
          int       33
          endm
;
get_string  macro   limit,string        ;GET_STRING
          mov       string,limit
          mov       dx,offset string
          mov       ah,10               ;function 10
          int       33
          endm
;
check_kbd_status macro                  ;CHECK_KBD_STATUS
          mov       ah,11               ;function 11
          int       33
          endm
;
flush_and_read_kbd  macro  switch       ;FLUSH_AND_READ_KBD
```

```
            mov       al,switch
            mov       ah,12                  ;function 12
            int       33
            endm
;
reset_disk macro                             ;RESET DISK
            mov       ah,13                  ;function 13
            int       33
            endm
;;page
select_disk  macro    disk                   ;SELECT_DISK
            mov       dl,disk[-65]
            mov       ah,14                  ;function 14
            int       33
            endm
;
open        macro     fcb                    ;OPEN
            mov       dx,offset fcb
            mov       ah,15                  ;function 15
            int       33
            endm
;
close       macro     fcb                    ;CLOSE
            mov       dx,offset fcb
            mov       ah,16                  ;function 16
            int       33
            endm
;
search_first macro    fcb                    ;SEARCH_FIRST
            mov       dx,offset fcb
            mov       ah,17                  ;Function 17
            int       33
            endm
;
search_next  macro    fcb                    ;SEARCH_NEXT
            mov       dx,offset fcb
            mov       ah,18                  ;function 18
            int       33
            endm
;
delete      macro     fcb                    ;DELETE
            mov       dx,offset fcb
            mov       ah,19                  ;function 19
            int       33
            endm
;
read_seq    macro     fcb                    ;READ_SEQ
            mov       dx,offset fcb
            mov       ah,20                  ;function 20
            int       33
            endm
;
write_seq macro       fcb                    ;WRITE_SEQ
            mov       dx,offset fcb
            mov       ah,21                  ;function 21
```

```
           int       33
           endm
;
create     macro     fcb                    ;CREATE
           mov       dx,offset fcb
           mov       ah,22                  ;function 22
           int       33
           endm
;
rename     macro     fcb,newname            ;RENAME
           mov       dx,offset fcb
           mov       ah,23                  ;function 23
           int       33
           endm
;
current_disk macro                          ;CURRENT_DISK
           mov       ah,25                  ;function 25
           int       33
           endm
;
set_dta    macro     buffer                 ;SET_DTA
           mov       dx,offset buffer
           mov       ah,26                  ;function 26
           int       33
           endm
;
alloc_table macro                           ;ALLOC_TABLE
           mov       ah,27                  ;function 27
           int       33
           endm
;
read_ran   macro     fcb                    ;READ_RAN
           mov       dx,offset fcb
           mov       ah,33                  ;function 33
           int       33
           endm
;
write_ran macro      fcb                    ;WRITE_RAN
           mov       dx,offset fcb
           mov       ah,34                  ;function 34
           int       33
           endm
;
file_size macro      fcb                    ;FILE_SIZE
           mov       dx,offset fcb
           mov       ah,35                  ;function 35
           int       33
           endm
;
set_relative_record  macro  fcb             ;SET_RELATIVE_RECORD
           mov       dx,offset fcb
           mov       ah,36                  ;function 36
   int       33
           endm
;;page
```

```
set_vector  macro  interrupt,seg_addr,off_addr  ;SET_VECTOR
          push      ds
          mov       ax,seg_addr
          mov       ds,ax
          mov       dx,off_addr
          mov       al,interrupt
          mov       ah,37              ;function 37
          int       33
          endm
;
create_prog_seg  macro  seg_addr       ;CREATE_PROG_SEG
          mov       dx,seg_addr
          mov       ah,38              ;function 38
          int       33
          endm
;
ran_block_read  macro  fcb,count,rec_size ;RAN_BLOCK_READ
          mov       dx,offset fcb
          mov       cx,count
          mov       word ptr fcb[14],rec_size
          mov       ah,39              ;function 39
          int       33
          endm
;
ran_block_write  macro  fcb,count,rec_size ;RAN_BLOCK_WRITE
          mov       dx,offset fcb
          mov       cx,count
          mov       word ptr fcb[14],rec_size
          mov       ah,40                     ;function 40
          int       33
          endm
;
parse     macro     filename,fcb              ;PARSE
          mov       si,offset filename
          mov       di,offset fcb
          push      es
          push      ds
          pop       es
          mov       al,15
          mov       ah,41                     ;function 41
          int       33
          pop       es
          endm
;
get_date  macro                               ;GET_DATE
          mov       ah,42                     ;function 42
          int       33
          endm
;;page
set_date  macro     year,month,day            ;SET_DATE
          mov       cx,year
          mov       dh,month
          mov       dl,day
          mov       ah,43                     ;function 43
          int       33
```

```
          endm
;
get_time  macro                                   ;GET_TIME
          mov       ah,44                         ;function 44
          int       33
          endm
;
                                                  ;SET_TIME
set_time  macro     hour,minutes,seconds,hundredths
          mov       ch,hour
          mov       cl,minutes
          mov       dh,seconds
          mov       dl,hundredths
          mov       ah,45                         ;function 45
          int       33
          endm
;
verify    macro     switch                        ;VERIFY
          mov       al,switch
          mov       ah,46                         ;function 46
          int       33
          endm
;
;*******************
; General
;*******************
;
move_string  macro  source,destination,num_bytes
                                   ;MOVE_STRING
          push      es
          mov       ax,ds
          mov       es,ax
          assume    es:data
          mov       si,offset source
          mov       di,offset destination
          mov       cx,num_bytes
      rep movs      es:destination,source
          assume    es:nothing
          pop       es
          endm
;
;
convert   macro     value,base,destination        ;CONVERT
          local     table,start
          jmp       start
table     db        "0123456789ABCDEF"
start:    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
            endm
;;page
convert_to_binary  macro  string,number,value
                                 ;CONVERT_TO_BINARY
            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[25]
            mov       cl,5
            shr       dl,cl
            mov       dh,dir_entry[25]
            and       dh,1fh
            xor       cx,cx
            mov       cl,dir_entry[26]
            shr       cl,1
            add       cx,1980
            endm
```

1.9 EXTENDED EXAMPLE OF MS-DOS SYSTEM CALLS

```
title DISK DUMP
zero                        equ  0
disk_B                      equ  1
sectors_per_read            equ  9
cr                          equ  13
blank                       equ  32
period                      equ  46
tilde                       equ  126
      INCLUDE  B:CALLS.EQU
;
subttl DATA SEGMENT
page +
data                        segment
;
input_buffer                db   9 dup(512 dup(?))
output_buffer               db   77 dup(" ")
                            db   0DH,0AH,"$"
start_prompt                db   "Start at sector: $"
sectors_prompt              db   "Number of sectors: $"
continue_prompt             db   "RETURN to continue $"
header                      db   "Relative sector $"
end_string                  db   0DH,0AH,0AH,07H,"ALL DONE$"
                                 ;DELETE THIS
crlf                        db   0DH,0AH,"$"
table                       db   "0123456789ABCDEF$"
;
ten                         db   10
sixteen                     db   16
;
start_sector                dw   1
sector_num               label   byte
sector_number               dw   0
sectors_to_dump             dw   sectors_per_read
sectors_read                dw   0
;
buffer                   label   byte
max_length                  db   0
current_length              db   0
digits                      db   5 dup(?)
;
data                        ends
;
subttl STACK SEGMENT
page +
stack                       segment   stack
                            dw        100 dup(?)
stack_top                   label     word
stack                       ends
;
subttl MACROS
page +
;
```

```
        INCLUDE B:CALLS.MAC
;BLANK LINE
blank_line              macro     number
                        local     print_it
                        push      cx
                        call      clear_line
                        mov       cx,number
print_it:               display   output_buffer
                        loop      print_it
                        pop       cx
                        endm
;
subttl ADDRESSABILITY
page +
code                    segment
                        assume    cs:code,ds:data,ss:stack
start:                  mov       ax,data
                        mov       ds,ax
                        mov       ax,stack
                        mov       ss,ax
                        mov       sp,offset stack_top
;
                        jmp       main_procedure
subttl PROCEDURES
page +
;
;  PROCEDURES
;  READ DISK
read_disk               proc;
                        cmp       sectors_to_dump,zero
                        jle       done
                        mov       bx,offset input_buffer
                        mov       dx,start_sector
                        mov       al,disk_b
                        mov       cx,sectors_per_read
                        cmp       cx,sectors_to_dump
                        jle       get_sector
                        mov       cx,sectors_to_dump
get_sector:             push      cx
                        int       disk_read
                        popf
                        pop       cx
                        sub       sectors_to_dump,cx
                        add       start_sector,cx
                        mov       sectors_read,cx
                        xor       si,si
done:                   ret
read_disk               endp
;CLEAR_LINE
clear_line              proc;
                        push      cx
                        mov       cx,77
                        xor       bx,bx
move_blank:             mov       output_buffer[bx],' '
                        inc       bx
```

```
                        loop      move_blank
                        pop       cx
                        ret
clear line              endp
;
;PUT_BLANK
put_blank               proc;
                        mov       output_buffer[di]," "
                        inc       di
                        ret
put_blank               endp
;
;
setup                   proc;
                        display   start_prompt
                        get_string  4,buffer
                        display   crlf
                        convert_to_binary  digits,
                        current_length,start_sector
                        mov       ax,start_sector
                        mov       sector_number,ax
                        display   sectors_prompt
                        get_string  4,buffer
                        convert_to_binary  digits,
                        current_length,sectors_to_dump
                        ret
setup                   endp
;
;CONVERT_LINE
convert_line            proc;
                        push      cx
                        mov       di,9
                        mov       cx,16
convert_it:             convert   input_buffer[si],sixteen,
                        output_buffer[di]
                        inc       si
                        add       di,2
                        call      put_blank
                        loop      convert_it
                        sub       si,16
                        mov       cx,16
                        add       di,4
display_ascii:          mov       output_buffer[di],period
                        cmp       input_buffer[si],blank
                        jl        non_printable
                        cmp       input_buffer[si],tilde
                        jg        non_printable
printable:              mov       dl,input_buffer[si]
                        mov       output_buffer[di],dl
non_printable:          inc       si
                        inc       di
                        loop      display_ascii
                        pop       cx
                        ret
convert_line            endp
```

```
;
;DISPLAY_SCREEN
display_screen          proc;
                        push      cx
                        call      clear_line
;
                        mov       cx,17
;I WANT length header
                        dec       cx
;minus 1 in cx
                        xor       di,di
move_header:            mov       al,header[di]
                        mov       output_buffer[di],al
                        inc       di
                        loop      move_header    ;FIX THIS!
;
                        convert   sector_num[1],sixteen,
                        output_buffer[di]
                        add       di,2
                        convert   sector_num,sixteen,
                        output_buffer[di]
                        display   output_buffer
                        blank_line 2
                        mov       cx,16
dump_it:                call      clear_line
                        call      convert_line
                        display   output_buffer
                        loop      dump_it
                        blank_line 3
                        display   continue_prompt
                        get_char_no_echo
                        display   crlf
                        pop       cx
                        ret
display_screen          endp
;
;
;   END PROCEDURES
subttl MAIN PROCEDURE
page +
main_procedure:         call      setup
check_done:             cmp       sectors_to_dump,zero
                        jng       all_done
                        call      read_disk
                        mov       cx,sectors_read
display_it:             call      display_screen
                        call      display_screen
                        inc       sector_number
                        loop      display_it
                        jmp       check_done
all_done:               display   end_string
                        get_char_no_echo
code                    ends
                        end       start
```

CHAPTER 2

MS-DOS 2.0 DEVICE DRIVERS

2.1 WHAT IS A DEVICE DRIVER?

A device driver is a binary file with all of the code in it to manipulate the hardware and provide a consistent interface to MS-DOS. In addition, it has a special header at the beginning that identifies it as a device, defines the strategy and interrupt entry points, and describes 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 are designed to perform serial character I/O like CON, AUX, and PRN. These devices are named (i.e., CON, AUX, CLOCK, etc.), and users 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 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 are identified via the drive letters (A:, B:, C:, etc.).

Block devices also have units. A single driver may be responsible for one or more 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 (0-3) defined 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. MS-DOS 2.0 is not limited to 16 block device units, as previous versions were. The theoretical limit is 63 (26 - 1), but it should be noted that after 26 the drive letters are unconventional (such as], \, and ^).

NOTE

Character devices cannot define multiple units because they have only one name.

2.2 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               |
| (Must be set to -1)                        |
+--------------------------------------------+
| WORD attributes                            |
|  Bit 15 = 1 if char device 0 is blk        |
|  if bit 15 is 1                            |
|      Bit 0 = 1 if current sti device       |
|      Bit 1 = 1 if current sto output       |
|      Bit 2 = 1 if current NUL device       |
|      Bit 3 = 1 if current CLOCK dev        |
|      Bit 4 = 1 if special                  |
|      Bits 5-12 Reserved; must be set       |
|                to 0                        |
|  Bit 14 is the IOCTL bit                   |
|  Bit 13 is the NON IBM FORMAT bit          |
+--------------------------------------------+
| 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. 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.

2.2.1 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 .COM file, the last driver in the file must have the pointer to the next Device Header field set to -1.

2.2.2 Attribute Field

The attribute field is used to tell the system whether this device is a block or character device (bit 15). Most other bits are used to give selected character devices certain special treatment. (Note that these bits mean nothing on a block device). For example, assume that a user has a new device driver that he wants to be the standard input and output. Besides 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 2.7, "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 IBM FORMAT bit applies only to block devices and affects the operation of the BUILD BPB (Bios Parameter Block) device call. (Refer to Section 2.5.3, "MEDIA CHECK and BUILD BPB," for further information on this call).

The other bit of interest is the IOCTL bit, which has meaning on character and block devices. 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 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.

2.2.3 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.

2.2.4 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" in this chapter for more information.

2.3 HOW TO CREATE A DEVICE DRIVER

In order to create a device driver that MS-DOS can install, you must write a binary file with a Device Header at the beginning of the file. Note that for device drivers, the code should not be originated at 100H, but rather at 0. The link field (pointer to next Device Header) 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.

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.

NOTE

Because MS-DOS can install the driver anywhere in memory, care must be taken in any far memory references. You should not expect that your driver will always be loaded in the same place every time.

2.4 INSTALLATION OF DEVICE DRIVERS

MS-DOS 2.0 allows new device drivers to be installed dynamically at boot time. This is accomplished by INIT code in the BIOS, which reads and processes the CONFIG.SYS file.

MS-DOS calls upon the device drivers to perform their function in the following manner:

> MS-DOS makes a far call to strategy entry, and passes (in a Request Header) the information describing the functions of the device driver.

This structure allows you to program an interrupt-driven device driver. For example, you may want to perform local buffering in a printer.

2.5 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 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 3. Request Header

2.5.1 Unit Code

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.

2.5.2 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 only, NOP for character)
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)

2.5.3 MEDIA CHECK And BUILD BPB

MEDIA CHECK and BUILD BPB are used with block devices only.

MS-DOS calls MEDIA CHECK first for a drive unit. MS-DOS passes its current media descriptor byte (refer to the section "Media Descriptor Byte" later in this chapter). MEDIA CHECK returns one of the following results:

Media Not Changed - current DPB and media byte are OK.

Media Changed - Current DPB and media are wrong. MS-DOS invalidates any buffers for this unit and calls the device driver to build the BPB with media byte and buffer.

Not Sure - If there are dirty buffers (buffers with changed data, not yet written to disk) for this unit, MS-DOS assumes the DPB and media byte are OK (media not changed). If nothing is dirty, MS-DOS assumes the media has changed. It invalidates any buffers for the unit, and calls the device driver to build the BPB with media byte and buffer.

Error - If an error occurs, MS-DOS sets the error code accordingly.

MS-DOS will call BUILD BPB under the following conditions:

If Media Changed is returned

If Not Sure is returned, and there are no dirty buffers

The BUILD BPB call also gets a pointer to a one-sector buffer. What this buffer contains is determined by the NON IBM FORMAT bit in the attribute field. If the bit is zero (device is IBM format-compatible), then the buffer contains the first sector of the first FAT. The FAT ID byte is the first byte of this buffer. NOTE: The BPB must be the same, as far as location of the FAT is concerned, for all possible media because this first FAT sector must be read <u>before</u> the actual BPB is returned. If the NON IBM FORMAT bit is set, then the pointer points to one sector of scratch space (which may be used for anything).

2.5.4 Status Word

The following figure illustrates the status word 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	D O N	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 is complete. For MS-DOS 2.0, 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

Bit 9 is the busy bit, which is set only by status calls.

<u>For output on character devices</u>: If bit 9 is 1 on return, 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>: If bit 9 is 1 on return, 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 return quickly. It also indicates that something has been typed. MS-DOS assumes 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 MS-DOS will not continuously wait for something to get into a buffer that does not exist.

One of the functions defined for each device is INIT. This routine is called only once when the device is installed. The INIT routine returns a location (DS:DX), which is a pointer to the first free byte of memory after the device driver (similar to "Keep Process"). This pointer method can be used to delete initialization code that is only needed once, saving on space.

Block devices are installed the same way and also return a first free byte pointer as described above. Additional information is also returned:

The number of units is returned. This determines 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).

A pointer to a BPB (BIOS Parameter Block) pointer array is also returned. There is one table for each unit defined. These blocks will be used to build an internal DOS data structure for each of the units. The pointer passed to the DOS from the driver points to an array of n word pointers to BPBs, where n is the number of units defined. In this way, if all units are the same, all of the pointers can point to the same BPB, saving space. Note that this array 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 sector size defined must be less than or equal to the maximum sector size defined at default BIOS INIT time. If it isn't, the install will fail.

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 take several approaches; they may be 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 side, unit 1 = drive 0 double side. 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 at INIT must define space large enough 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.

2.6 FUNCTION CALL PARAMETERS

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.

NOTE

All DWORD pointers are stored offset first, then segment.

2.6.1 INIT

Command code = 0

INIT - ES:BX ->

13-BYTE Request Header
BYTE # of units
DWORD break address
DWORD pointer to BPB array (Not set by character devices)

The number of units, break address, and BPB pointer are set by the driver. On entry, the DWORD that is to be set to the BPB array (on block devices) points to the character after the '=' on the line in CONFIG.SYS that loaded this device. This allows drivers to scan the CONFIG.SYS invocation line for arguments.

NOTE

If there are multiple device drivers in a single .COM 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 .COM file return the same ending address.

2.6.2 MEDIA CHECK

Command Code = 1

MEDIA CHECK - ES:BX ->

13-BYTE Request Header
BYTE media descriptor from DPB
BYTE returned

In addition to setting the status word, the driver must set the return byte to one of the following:

```
-1 Media has been changed
 0 Don't know if media has been changed
 1 Media has not been changed
```

If the driver can return -1 or 1 (by having a door-lock or other interlock mechanism) MS-DOS performance is enhanced because MS-DOS does not need to reread the FAT for each directory access.

2.6.3 BUILD BPB (BIOS Parameter Block)

Command code = 2

BUILD BPB - ES:BX ->

```
+-------------------------------------+
| 13-BYTE Request Header              |
+-------------------------------------+
| BYTE media descriptor from DPB      |
+-------------------------------------+
| DWORD transfer address              |
| (Points to one sector worth of      |
|  scratch space or first sector      |
|  of FAT depending on the value      |
|  of the NON IBM FORMAT bit)         |
+-------------------------------------+
| DWORD pointer to BPB                |
+-------------------------------------+
```

If the NON IBM FORMAT bit of the device is set, then the DWORD transfer address points to a one sector buffer, which can be used for any purpose. If the NON IBM FORMAT bit is 0, then this buffer contains the first sector of the first FAT and the driver must not alter this buffer.

If IBM compatible format is used (NON IBM FORMAT BIT = 0), then the first sector of the first FAT must be located at the same sector on all possible media. This is because the FAT sector will be read BEFORE the media is actually determined. Use this mode if all you want is to read the FAT ID byte.

In addition to setting status word, the driver must set the Pointer to the BPB on return.

In order to allow for many different OEMs to read each other's disks, the following standard is suggested: The information relating to the BPB for a particular piece of media is kept in the boot sector for the media. In particular, the format of the boot sector is:

	3 BYTE near JUMP to boot code
	8 BYTES OEM name and version
B	WORD bytes per sector
P 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

The three words at the end (sectors per track, number of heads, and number of hidden sectors) are optional. They are intended to help the BIOS understand the media. Sectors per track may be redundant (could be calculated from total size of the disk). Number of heads is useful for supporting different multi-head drives which have the same storage capacity, but different numbers of surfaces. Number of hidden sectors may be used to support drive-partitioning schemes.

2.6.4 Media Descriptor Byte

The last two digits of the FAT ID byte are called the media descriptor byte. Currently, the media descriptor byte has been defined for a few media types, including 5-1/4" and 8" standard disks. For more information, refer to Section 3.6, "MS-DOS Standard Disk Formats."

Although these media bytes map directly to FAT ID bytes (which are constrained to the 8 values F8-FF), media bytes can, in general, be any value in the range 0-FF.

2.6.5 READ Or WRITE

Command codes = 3,4,8,9, and 12

READ or WRITE - ES:BX (Including IOCTL) ->

13-BYTE Request Header
BYTE media descriptor from DPB
DWORD transfer address
WORD byte/sector count
WORD starting sector number (Ignored on character devices)

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 driver must correctly set the return sector (byte) count to the actual number of bytes transferred.

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.

2.6.6 NON DESTRUCTIVE READ NO WAIT

Command code = 5

NON DESRUCTIVE READ NO WAIT - ES:BX ->

13-BYTE Request Header
BYTE read from device

If the character device returns busy bit = 0 (characters in 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"). Basically, this call allows MS-DOS to look ahead one input character.

2.6.7 STATUS

Command codes = 6 and 10

STATUS Calls - ES:BX ->

13-BYTE Request Header

All the driver must do is set the status word and the busy bit as follows:

<u>For output on character devices</u>: If bit 9 is 1 on return, 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 means, a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the devices buffer and a read would return quickly. A return of 0 also indicates 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 buffer which doesn't exist.

2.6.8 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.

2.7 THE CLOCK DEVICE

One of the most popular add-on boards is the real time clock board. To allow this board to be integrated into the system for TIME and DATE, there is a special device (determined by the attribute word) called the CLOCK device. The CLOCK device defines and performs functions like any other character device. Most functions will be: "set done bit, reset error bit, return." When a read or write to this device occurs, exactly 6 bytes are transferred. The first two bytes are a word, which is the count of days since 1-1-80. The third byte is minutes; the fourth, hours; the fifth, hundredths of seconds; and the sixth, seconds. Reading the CLOCK device gets the date and time; writing to it sets the date and time.

2.8 EXAMPLE OF DEVICE DRIVERS

The following examples illustrate a block device driver and a character device driver program.

2.8.1 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,11
        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    GETBP                          ;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    GETBP                   ;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
                                            ;lunknown
        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 transfe
        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 GETBP 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 LOWER CASE
        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
```

2.8.2 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

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 3 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 (Fatal Error Abort 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 4BH) 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
Second copy of file allocation table - variable size(optional)
Additional copies of file allocation table-variable size (opt.)
Root directory - variable size
File data area

Allocation of 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; all of the clusters for a file are "chained" together in the File Allocation Table (FAT). (Refer to Section 3.5, "File Allocation Table.") There is usually a second copy of the FAT kept, for consistency. 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 unusable disk.

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.

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 File 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 a 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 sub-directory, 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 Change 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:

```
Offset 17H
| H | H | H | H | H | M | M | M |
  7               3   2

Offset 16H
| M | M | M | S | S | S | S | S |
          5   4               0
```

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 |
 7                        1   0

Offset 18H
| M | M | M | D | D | D | D | D |
          5   4               0
```

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 Section 3.5.1, "How to Use the File Allocation Table," 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. The driver is then responsible for locating the logical sector on disk. Programs must 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. The first two FAT entries map a portion of the directory; these FAT entries indicate the size and format of the disk.

The second and third bytes currently always contain FFH.

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 found 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 hexadecimal characters:

000	If the cluster is unused and available.
FF7	The cluster has a bad sector in it. 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.
FF8-FFF	Indicates the last cluster of a file.
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 section after the reserved sectors. If the FAT is larger than one sector, the sectors are continguous. 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 File Allocation Table

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.

3.6 MS-DOS STANDARD DISK FORMATS

On an MS-DOS disk, the clusters are 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.

For disks, the following table can be used:

# Sides	Sectors/ Track	FAT size Sectors	Dir Sectors	Dir Entries	Sectors/ Cluster
1	8	1	4	64	1
2	8	1	7	112	2
1	9	2	4	64	1
2	9	2	7	112	2

Figure 4. 5-1/4" Disk Format

The first byte of the FAT can sometimes be used to determine the format of the disk. The following 5-1/4" formats have been defined for the IBM Personal Computer, based on values of the first byte of the FAT. The formats in Table 3.1 are considered to be the standard disk formats for MS-DOS.

Table 3.1 MS-DOS Standard Disk Formats

	5-1/4	5-1/4	5-1/4	5-1/4	8	8	8
No. sides	1	1	2	2	1	1	2
Tracks/side	40	40	40	40	77	77	77
Bytes/ sector	512	512	512	512	128	128	1024
Sectors/ track	8	9	8	9	26	26	8
Sectors/allo- cation unit	1	1	2	2	4	4	1
Reserved sectors	1	1	1	1	1	4	1
No. FATs	2	2	2	2	2	2	2
Root directory entries	64	64	112	112	68	68	192
No. sectors	320	360	640	720	2002	2002	616
Media Descriptor Byte	FE	FC	FF	FD	FE*	FD	FE*
Sectors for 1 FAT	1	2	1	2	6	6	2

*The two media descriptor bytes that are the same for 8" disks (FEH) is not a misprint. To establish whether a disk is single- or double-density, a read of a single-density address mark should be made. If an error occurs, the media is double-density.

CHAPTER 4

MS-DOS CONTROL BLOCKS AND WORK AREAS

4.1 TYPICAL MS-DOS MEMORY MAP

0000:0000	Interrupt vector table
XXXX:0000	IO.SYS - MS-DOS interface to hardware
XXXX:0000	MSDOS.SYS - MS-DOS interrupt handlers, service routines (Interrupt 21H functions)
	MS-DOS buffers, control areas, and installed device drivers
XXXX:0000	Resident part of COMMAND.COM - Interrupt handlers for Interrupts 22H (Terminate Address), 23H (CONTROL-C Exit Address), 24H (Fatal Error Abort Address) and code to reload the transient part
XXXX:0000	External command or utility - (.COM or .EXE file)
XXXX:0000	User stack for .COM files (256 bytes)
XXXX:0000	Transient part of COMMAND.COM - Command interpreter, internal commands, batch processor

1. Memory map addresses are in segment:offset format. For example, 0090:0000 is absolute address 0900H.

2. 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 four methods:

1. A long jump to offset 0 in the Program Segment Prefix

2. By issuing an INT 20H with CS:0 pointing at the PSP

3. By issuing an INT 21H with register AH=0 with CS:0 pointing at the PSP, or 4CH and no restrictions on CS

4. By a long call to location 50H in the Program Segment Prefix with AH=0 or Function Request 4CH

NOTE

> It is the responsibility of all programs to ensure that the CS register contains the segment address of the Program Segment Prefix when terminating via any of these methods, except Function Request 4CH. For this reason, using Function Request 4CH is the preferred method.

All four methods result in transferring control to the program that issued the EXEC. During this returning process, Interrupts 22H, 23H, and 24H (Terminate Address, CONTROL-C Exit Address, and Fatal Error Abort 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:

<u>For all programs</u>:

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. 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.

Offset 50H in the Program Segment Prefix contains code to call the MS-DOS function dispatcher. By placing the desired function request number in AH, a program can issue a far call to offset 50H to invoke an MS-DOS function, rather than issuing an Interrupt 21H. Since this is a call and not an interrupt, MS-DOS may place any code appropriate to making a system call at this position. This makes the process of calling the system portable.

The Disk Transfer Address (DTA) is set to 80H (default DTA in the Program Segment Prefix).

File control blocks at 5CH and 6CH 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 unavailable 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).

For Executable (.EXE) programs:

DS and ES registers are set to point to the Program Segment Prefix.

CS,IP,SS, and SP registers are set to the values passed by MS-LINK.

For Executable (.COM) programs:

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 5 illustrates the format of the Program Segment Prefix. All offsets are in hexadecimal.

(offsets in hex)

Offset				
0	INT 20H	End of alloc. block*	Reserved	Long call to MS-DOS function dispatcher (5 bytes)**
8		Terminate address (IP, CS)		CTRL-C exit address (IP)
10	CTRL-C exit address (CS)	Hard error exit address (IP, CS)		
	Used by MS-DOS *** 2CH 5CH			
	Formatted Parameter Area 1 formatted as standard unopened FCB 6CH			
	Formatted Parameter Area 2 formatted as standard unopened FCB (overlaid if FCB at 5CH is opened)			
80	Unformatted Parameter Area (default Disk Transfer Area)			
100				

Figure 5. 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

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 4BH, Load and Execute a Program, to load (or load and execute) an .EXE file.

The .EXE files produced by 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.

INDEX

MICROSOFT™

10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name ____________________

Street ____________________

City ____________ State ________ Zip ________

Phone ____________ Date ________

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem (Document #__________)

_____ Other

Software Description

Microsoft Product ____________________

Rev. ________ Registration # ____________

Operating System ____________________

Rev. ________ Supplier ____________

Other Software Used ____________________

Rev. ________ Supplier ____________

Hardware Description

Manufacturer ____________ CPU ________ Memory ______ KB

Disk Size ______″ Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals ____________________

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support ____________ Date Received ____________

Routing Code ____________ Date Resolved ____________

Report Number ____________

Action Taken:

Part no.: SPR00

MICROSOFT™