NCR

NCR DECISION MATE V

MS™- DOS
Programmer's Manual

MACRO-86, MS-CREF, MS-LINK, MS-LIB, and MS-DOS (and its constituent program names EDLIN and DEBUG are trademarks of Microsoft Corporation. Microsoft is a registered trademark of Microsoft Corporation.

Copyright © 1983, 1984 by NCR Corporation
Dayton, Ohio
All Rights Reserved
Printed in the Federal Republic of Germany

Third Edition, August 1984

It is the policy of NCR Corporation to improve products as new technology, componentes, software, and firmware become available. NCR Corporation, therefore, reserves the right to change specifications without prior notice.

All features. functions, and operations described herein may not be marketed by NCR in all parts of the world. In some instances, photographs are of equipment prototypes. Therefore, before using this document, consult your nearest dealer or NCR office for information that is applicable and current.

General Introduction

Chapter 1 System Calls

1.1 Introduction 1-1
1.2 Programming Considerations 1-1
1.2.1 Calling From Macro Assembler 1-1
1.2.2 Calling From a High-Level Language 1-1
1.2.3 Returning Control to MS-DOS 1-2
1.2.4 Console and Printer Input/Output Calls . . . 1-3
1.2.5 Disk I/O System Calls 1-3
1.3 File Control Block (FCB) 1-3
1.3.1 Fields of the FCB 1-4
1.3.2 Extended FCB 1-6
1.3.3 Directory Entry 1-6
1.3.4 Fields of the FCB 1-7
1.4 System Call Descriptions 1-9
1.4.1 Programming Examples 1-10
1.5 Xenix-Compatible Calls 1-11
1.6 Interrupts 1-14
16H Keyboard Character Code Read 1-16
20H Program Terminate 1-17
21H Function Request 1-18
22H Terminate Address 1-19
23H CONTROL-C Exit Address 1-19
24H Fatal Error Abort Address 1-20
25H Absolute Disk Read 1-23
26H Absolute Disk Write 1-25
27H Terminate But Stay Resident 1-27
1.7 Function Requests 1-28
1.7.1 CP/M-Compatible Calling Sequence 1-28
1.7.2 Treatment of Registers 1-29

Function Requests

00H Terminate Program 1-33
01H Read Keyboard and Echo 1-34
02H Display Character 1-35
03H Auxiliary Input 1-36
04H Auxiliary Output 1-37
05H Print Character 1-38
06H Direct Console I/O 1-40
07H Direct Console Input 1-42
08H Read Keyboard 1-43
09H Display String 1-44
0AH Buffered Keyboard Input 1-45
0BH Check Keyboard Status 1-47

0CH Flush Buffer, Read Keyboard 1-48
0DH Disk Reset 1-49
0EH Select Disk 1-50
0FH Open File 1-50
10H Close File 1-53
11H Search for First Entry 1-55
12H Search for Next Entry 1-57
13H Delete File 1-59
14H Sequential Read 1-61
15H Sequential Write 1-63
16H Create File 1-65
17H Rename File 1-67
19H Current Disk 1-69
1AH Set Disk Transfer Address 1-70
21H Random Read 1-72
22H Random Write 1-74
23H File Size 1-76
24H Set Relative Record 1-78
25H Set Vector 1-80
27H Random Block Read 1-81
28H Random Block Write 1-84
29H Parse File Name 1-87
2AH Get Date 1-90
2BH Set Date 1-92
2CH Get Time 1-94
2DH Set Time 1-95
2EH Set/Reset Verify Flag 1-97
2FH Get Disk Transfer Address 1-99
30H Get DOS Version Number 1-100
31H Keep Process 1-101
33H CONTROL-C Check 1-102
35H Get Interrupt Vector 1-104
36H Get Disk Free Space 1-105
38H Return Country-Dependent Information 1-106
39H Create Sub-Directory 1-109
3AH Remove a Directory Entry 1-110
3BH Change Current Directory 1-111
3CH Create a File 1-112
3DH Open a File 1-113
3EH Close a File Handle 1-115
3FH Read From File/Device 1-116
40H Write to a File/Device 1-117
41H Delete a Directory Entry 1-118

42H Move a File Pointer 1-119
43H Change Attributes 1-120
44H I/O Control for Devices 1-121
45H Duplicate a File Handle 1-125
46H Force a Duplicate of a Handle 1-126
47H Return Text of Current Directory . . 1-127
48H Allocate Memory 1-128
49H Free Allocated Memory 1-129
4AH Modify Allocated Memory Blocks . . 1-130
4BH Load and Execute a Program 1-131
4CH Terminate a Process 1-134
4DH Retrieve the Return Code of a Child 1-135
4EH Find Match File 1-136
4FH Step Through a Directory Matching Files 1-138
54H Return Current Setting of Verify . . 1-139
56H Move a Directory Entry 1-140
57H Get/Set Date/Time of File 1-141
1.8 Macro Definitions for MS-DOS System Call Examples (00H-57H) 1-142
1.9 Extended Example of MS-DOS System Calls 1-149
Chapter 2 MS-DOS Device Drivers
2.1 What is a Device Driver? 2-1
2.2 Device Headers 2-3
2.2.1 Pointer to Next Device Field 2-3
2.2.2 Attribute Field 2-4
2.2.3 Strategy and Interrupt Routines 2-5
2.2.4 Name Field 2-5
2.3 How to Create a Device Driver 2-5
2.4 Installation of Device Drivers 2-6
2.5 Request Header 2-6
2.5.1 Unit Code 2-7
2.5.2 Command Code Field 2-7
2.5.3 MEDIA CHECK and BUILD BPB 2-8
2.5.4 Status Word 2-9
2.6 Function Call Parameters 2-11
2.6.1 INIT 2-12
2.6.2 MEDIA CHECK 2-12
2.6.3 BUILD BPB 2-13
2.6.4 Media Descriptor Byte 2-15
2.6.5 READ OR WRITE 2-16
2.6.6 NON DESTRUCTIVE READ NO WAIT 2-17
2.6.7 STATUS 2-18
2.6.8 FLUSH 2-18

2.7 The CLOCK Device 2-19

Chapter 3 MS-DOS Technical Information
3.1 MS-DOS Initialization 3-1
3.2 The Command Processor 3-1
3.3 MS-DOS Disk Allocation 3-3
3.4 MS-DOS Disk Directory 3-3
3.5 File Allocation Table 3-7
3.5.1 How to Use the File Allocation Table 3-8
3.6 IBM 5 1/4" MS-DOS Disk Formats 3-9

Chapter 4 MS-DOS Control Blocks and Work Areas
4.1 Typical MS-DOS Memory Map 4-1
4.2 MS-DOS Program Segment 4-2

Chapter 5 EXE File Structure and Loading

Chapter 6 Special Features 6-1
6.1 Timer Interrupt Support 6-1
6.1.1 Basis Concepts of the Timer Interrupt Support 6-1
6.1.2 Initilaization 6-5
6.2 I/O Control Functions 6-6
6.2.1 How to Write the Selected Values 6-6
6.2.2 How to Check the Selected Values 6-7
6.2.3 Pattern of the "Console Flags" Byte 6-8

A. Keyboard Code Charts A-1

Index

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 system calls and interrupts (Chapter 1). Chapter 2, "MS-DOS Device Drivers" contains information on how to install your own device drivers on MS-DOS. Chapter 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 6 describes special features, such as the timer interrupt support and I/O control functions. Appendix A provides keyboard code charts.

The term "MS-DOS" in this manual refers to MS-DOS versions that are 2.0 or higher.

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.

Calling from Microsoft Basic: 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 20 H 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 **unopened** FCB is one that contains only a drive specifier and filename, which can contain wild card characters (* and ?). An **opened** 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 ist 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

	Size	Offset	
Name	(bytes)	Hex	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
10H	0001 0000	16	Directory
16H	0001 0110	22	Hard attributes for FINDENTRY
20H	0010 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 repuired 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 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 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 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
16H	22	Keyboard Character Code Read
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

	Interrupt	
Description	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
Keyboard Character Code Read	16H	22
Program Terminate	20H	32
RESERVED - DO NOT USE	28-40H	40-64
Terminate Address	22H	34
Terminate But Stay Resident	27H	39

Keyboard Character Code Read (Interrupt 16H)

1. Normal Read

   Call
   AH=00H

   Return
   AH=AL
   Character code from keyboard

2. Non-destructive Read

   Call
   AH=01H

   Return
   AH=AL
   Character code from keyboard

   Zero flag set means there was not a character to get; Zero flag not set means AL and AH contain the character code from the keyboard.

Interrupt 16H allows keyboard read. 00 in register AH leads to a normal read – that means the program waits for a character to be typed, then returns it in AL and AH. 01 in Register AH leads to a non-destructive read, that is, the code read remains in the keyboard buffer.

NOTE

> Interrupt 16H gets the original keyboard codes (No translation to ASCII is made; the function keys are disabled). Turn to Appendix A for the US-English and International English + UK keyboard code charts. All registers except AX are preserved. There is no check for CONTROL-C.

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 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
Programm 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
          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 of 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 CS FLAGS	MS-DOS registers from issuing INT 24H
AX BX CX DX SI DI BP DS ES	User registers at time of original INT 21H request
IP CS FLAGS	From the original INT 21H from the user to MS-DOS

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 function 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             equu     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 lists 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/0
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 Info.
39H	Create Sub-Directory
3AH	Remove a Directory Entry
3BH	Change the 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/D 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 Info.	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 echoes 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 boths displays and prints characters as they are typed. If <NEW LINE> 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 1AH, 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 ne 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/0 (Function 06H)

Call
AH = 06H
DL
  See text

Return
AL
  If DL = FFH (255) before call, then Zero flag not set means AL has character from keyboard.
  Zero flag 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 **not** check for CONTROL-C.

Macro Definition:

```
dir-console-io macro    switch
               mov      dl,switch
               mov      ah,06H
               int      21H
               endm
```

Example

The following program sets the system clock to 0 and continuously displays the time. When any character is typed, the display 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 does not check for CONTROL-C, see Functions 01H or 07H.)

```
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-h	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 <NEW LINE> 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 <NEW LINE> is pressed. The string can be edited at 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 <NEW LINE> 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. (This is true only for MS-DOS versions that are higher than 2.0.)
> 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 = 10 H
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
AL
  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
AL
  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
             .
             .
fcb1         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     fcb1                ;THIS FUNCTION
             mov        fcb1[record-size],12
                                            ;set record size to 12
write-it:    write-seq fcb1                 ;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      fcb1                ;see Function 10H
```

Rename File (Function 17H)

Call
AH = 17H
DS:DX
  Modified FCB

Return
AL
  00H = Directory entry found
  EFH (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 09 H
           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      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-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      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-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 <NEW LINE>, the record is not replaced. 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      26 dup (?),13,10,"$"
prompt1         db      "Enter letter: $"
prompt2         db      "New record (<NEW LINE> for no change).$"
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
           push  ds
           mov   ax,seg-addr
           mov   ds,ax
           mov   dx,off-addr
           mov   ah,25H
           mov   al,interrupt
           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 32 K 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-string15,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-string15,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 equu    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-string15,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 20H).

Macro Definition:

```
parse       macro   string,fcb
            mov     si,offset string
            mov     di,offset fcb
            push    es
            push    ds
            pop     es
            mov     al,0FH ;bits 0, 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-string15,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          equu    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                ;coby 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 Disk Transfer Address.

Error returns:
None.

Example

```
        mov     ah,2FH
        int     21H
                ;es:bx has current Disk 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,30
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/0 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 codes 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
DS:DX
  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, a directory existed with the same name, or the path was not found.
3 = path not found
  The path specified had a syntax error.
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 pathname

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 pathname
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 send/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 Duplicat 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 = 47 H
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 48H.

Error return:
AX
  9 = invalid block
    The block passed in ES is not one allocated via Function Request 48H.
  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 configurations 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:XBIN;B:XBASICXLIB

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 Disk Transfer Address 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
            ; Disk Transfer 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 Disk Transfer 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

```
 ; Disk Transfer Address points at area returned by Function
   4EH
mov     ah, 4FH
int     21H
   ; next entry is at Disk Transfer Address
```

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 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
              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        macro    hour,minutes,seconds,hundredths ;SET-TIME
                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  destina-
                         tion
                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                   seg-
                        ment    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 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 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 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 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 **before** 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
ERR	RESERVED					BUS	DON	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 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.

> **For output on character devices:** 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.

> **For input on character devices with a buffer:** 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 Paramter 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 Paramter 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 P B ! V I B P B	WORD bytes per sector
	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
	BYTE media descriptor
	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:

> **For output on character devices:** 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.
> **For input on character devices with a buffer:** 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.

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 start-up, 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 (optional)
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 cirectory'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 18 H

```
| 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 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. 5-¼" Disk Format

The first byte of the FAT can sometimes be used to determine the format of the disk. The following 5-¼" 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 Flexible Disk Formats

	5-¼	5-¼	5-¼	5-¼	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/allocation 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.

Table 3.2 MS-DOS Standard Fixed Disk Formats

	5 MB*	5 MB	10 MB
BYTES PER SECTOR	512	512	512
SECTORS / ALLOCATION UNIT	16	16	8
RESERVED SECTORS	0	1	1
NUMBER OF FAT'S	1	1	2
ROOT DIRECTORY ENTRIES	512	496	512
NUMBER OF SECTORS PER DISK	10370	10370	20740
MEDIA DESCRIPTOR	FA	F9	F8
NUMBER OF FAT SECTORS	2	2	8
SECTORS PER TRACK	17	17	17
NUMBER OF HEADS	2	2	4
HIDDEN SECTORS	0	0	0

* The 5MB format was generated under D006-0052. All three formats can be used under D006-0225.

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 form 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:

For all programs:

The segment address of the passed environment is contained at offset 2CH in the Program Segment Prefix.

The environment is a series of ASCII strings (totaling less than 32K) in the form:

NAME = parameter

Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros. 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.

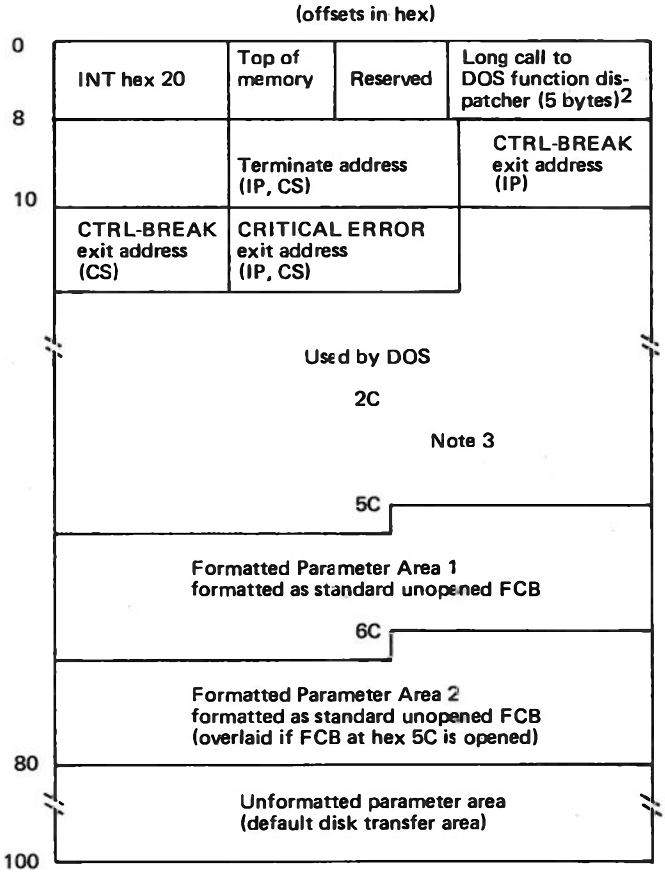
All of user memory is allocated to the program. If the program invokes another program through Function Request 4BH, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being executed.

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)
0
INT hex 20
Top of memory
Reserved
Long call to DOS function dispatcher (5 bytes)2
8
Terminate address (IP, CS)
CTRL-BREAK exit address (IP)
10
CTRL-BREAK exit address (CS)
CRITICAL ERROR exit address (IP, CS)
Used by DOS
2C
Note 3
5C
Formatted Parameter Area 1 formatted as standard unopened FCB
6C
Formatted Parameter Area 2 formatted as standard unopened FCB (overlaid if FCB at hex 5C is opened)
80
Unformatted parameter area (default disk transfer area)
100

1. First segment of available memory is in segment (paragraph) form (for example, hex 1000 would represent 64K).
2. The word at offset 6 contains the number of bytes available in the segment.
3. Offset hex 2C contains the segment address of the environment.

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 maxalloc are 0, then the program will be loaded as high as possible.
0E-0F	Initial value to be loaded into stack segment before starting program execution. This must be adjusted by relocation.
10-11	Value to be loaded into the SP register before starting program execution.
12-13	Negative sum of all the words in the file.
14-15	Initial value to be loaded into the IP register before starting program execution.
16-17	Initial value to be loaded into the CS register before starting program execution. This must be adjusted by relocation.
18-19	Relative byte offset from beginning of run file to relocation table.
1A-1B	The number of the overlay as generated by MS-LINK.

The relocation table follows the formatted area described above. This table consists of a variable number of relocation items. Each relocation item contains two fields: a two-byte offset value, followed by a two-byte segment value. These two fields contain the offset into the load module of a word which requires modification before the module is given control. The following steps describe this process:

1. The formatted part of the header is read into memory. Its size is 1BH.
2. A portion of memory is allocated depending on the size of the load module and the allocation numbers (0A-0B and 0C-0D). MS-DOS attempts to allocate FFFFH paragraphs. This will always fail, returning the size of the largest free block. If this block is smaller than minalloc and loadsize, then there will be no memory error. If this block is larger than maxalloc and loadsize, MS-DOS will allocate (maxalloc + loadsize). Otherwise, MS-DOS will allocate the largest free block of memory.
3. A Program Segment Prefix is built in the lowest part of the allocated memory.
4. The load module size is calculated by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted

based on the contents of offsets 02-03. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.

5. The load module is read into memory beginning with the start segment.
6. The relocation table items are read into a work area.
7. Each relocation table item segment value is added to the start segment value. This calculated segment, plus the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.
8. Once all relocation items have been processed, the SS and SP registers are set from the values in the header. Then, the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is the initial CS:IP to transfer to before starting execution of the program.

CHAPTER 6
SPECIAL FEATURES

6.1 TIMER INTERRUPT SUPPORT

The MS-DOS operating system provides support for an interval timer, which is used for updating DATE and TIME, on systems equipped with a 8088 16-Bit Processor with Interrupts (internal) 3273–K235. Applications and system modules can intercept the periodic interrupts of the interval timer for their own timing purposes.
Section 1 describes basic concepts, section 2 shows how timer and controller are initialized by MS-DOS.

6.1.1 Basic concepts of the timer interrupt support.

The interval timer, which is referred to as timer 2, is set by MS-DOS to issue an interrupt every 10 ms. Whenever a period of 10 ms expires, interrupt controller 8259A gives control to the timer 2 Interrupt Service Routine (= Interrupt type 8 vector). The ISR updates the internal counters and gives control to interrupt type/number 1CH vector.

As already mentioned, application programs can make use of the system's interval timer. To set up its own timer, an application program must save the interrupt number 1CH vector (address 0000:0070 hex). This can be done with function call 35H which returns the vector in ES:BS. The address in ES:BX should be saved as a Double WORD Pointer, because the application's timer ISR must exit to it.

The application must now set the vector to its own timer ISR with function call 25H. Interrupts must be disabled while this vector is being set. After this initial setup is accomplished, the application's timer ISR is given control every 10 ms.

To allow other application programs to make use of the interval timer too, every timer ISR must exit to the address saved during initial setup; it must not do an IRET.

NOTE

A timer ISR must always exit to the address saved during initial setup. If this is not the case, the system will crash.

Figures 1-3 shall illustrate the basic principles of an interval timer.

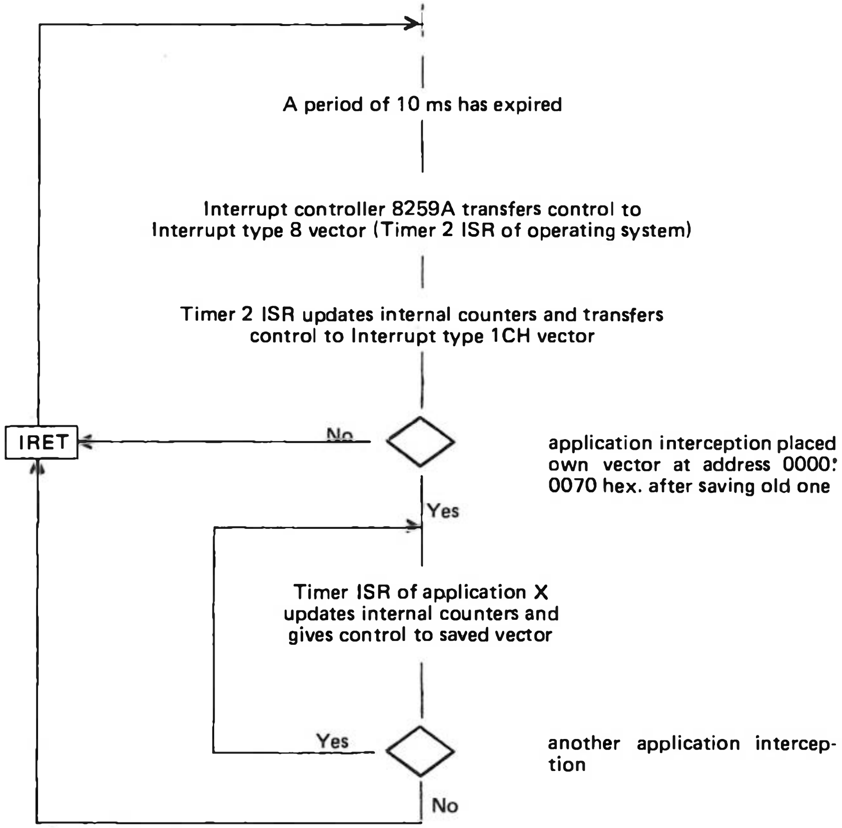

A period of 10 ms has expired
Interrupt controller 8259A transfers control to Interrupt type 8 vector (Timer 2 ISR of operating system)
Timer 2 ISR updates internal counters and transfers control to Interrupt type 1CH vector
IRET
No
application interception placed own vector at address 0000: 0070 hex. after saving old one
Yes
Timer ISR of application X updates internal counters and gives control to saved vector
Yes
another application interception
No

Figure 1 General Flow

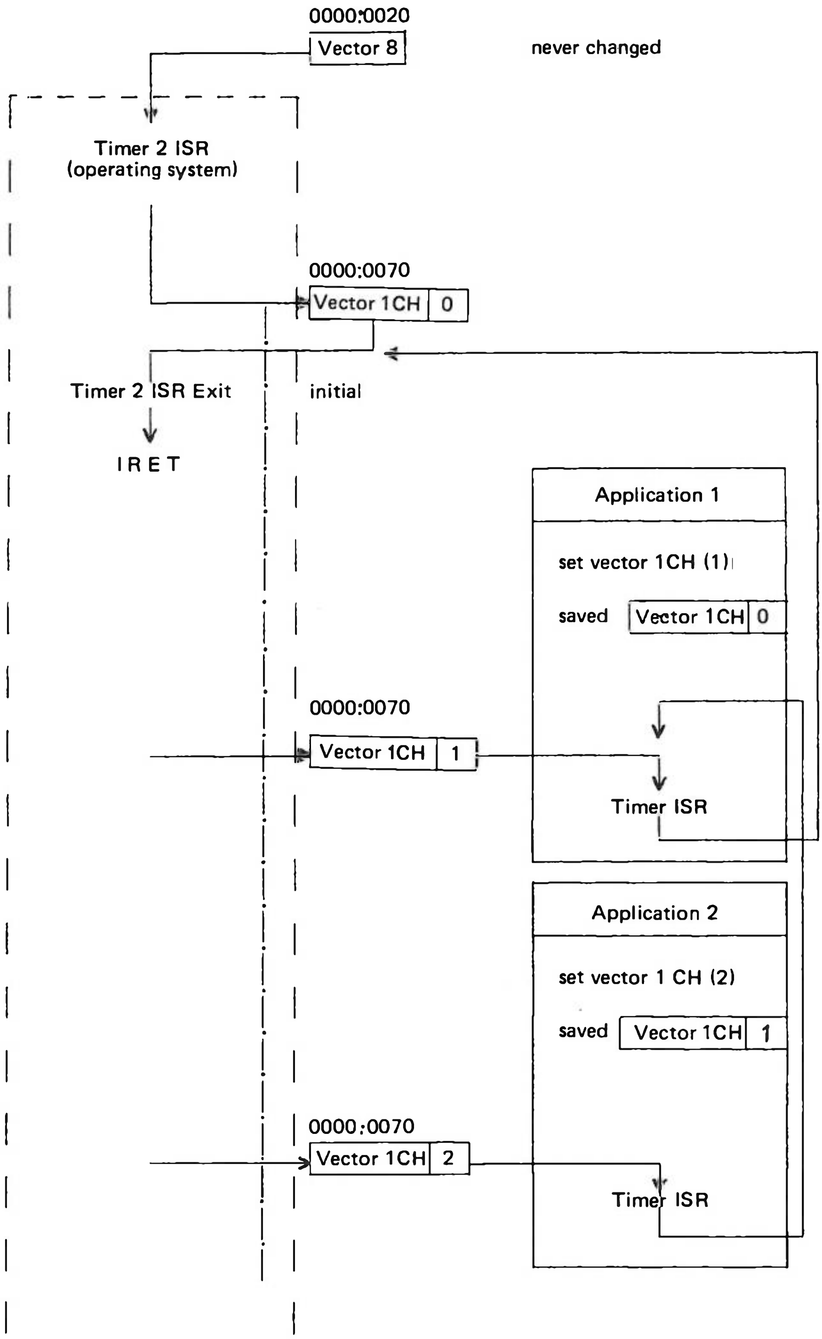

0000:0020
Vector 8
never changed
Timer 2 ISR
(operating system)
0000:0070
Vector 1CH
0
Timer 2 ISR Exit
initial
IRET
Application 1
set vector 1CH (1)
saved
Vector 1CH
0
0000:0070
Vector 1CH
1
Timer ISR
Application 2
set vector 1 CH (2)
saved
Vector 1CH
1
0000:0070
Vector 1CH
2
Timer ISR

Figure 2.

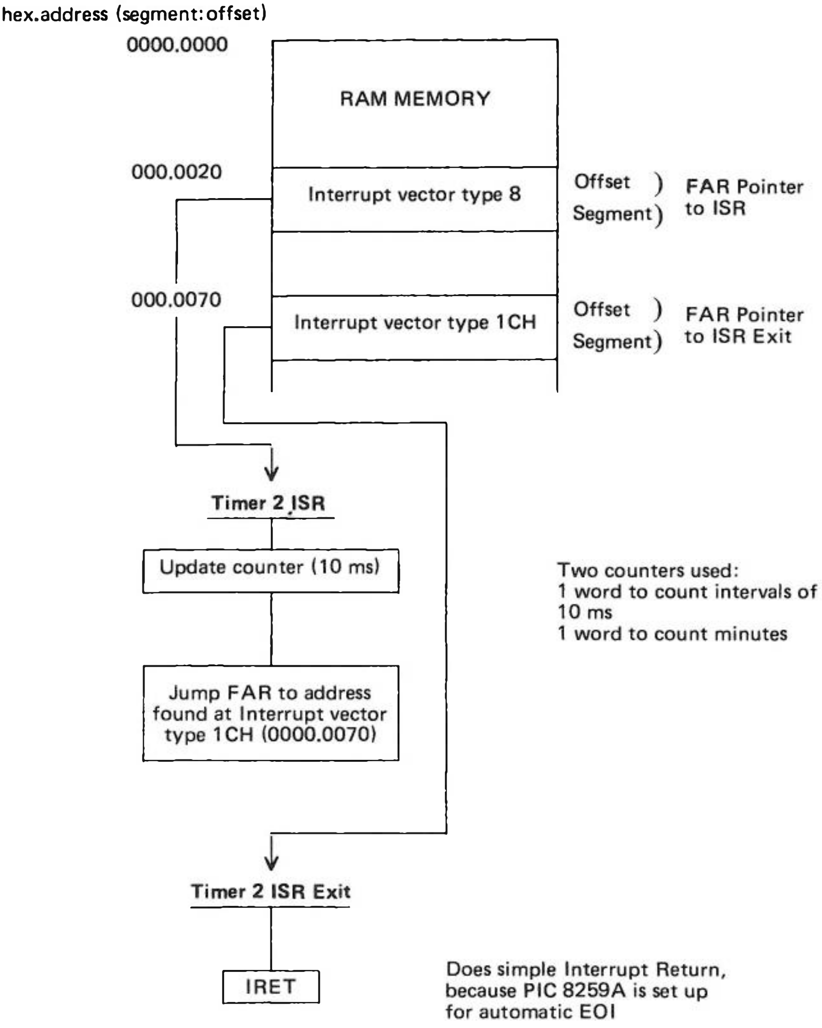

hex.address (segment:offset)
0000.0000
RAM MEMORY
000.0020
Interrupt vector type 8
Offset)
Segment)
FAR Pointer to ISR
000.0070
Interrupt vector type 1CH
Offset)
Segment)
FAR Pointer to ISR Exit
Timer 2 ISR
Update counter (10 ms)
Two counters used:
1 word to count intervals of 10 ms
1 word to count minutes
Jump FAR to address found at Interrupt vector type 1CH (0000.0070)
Timer 2 ISR Exit
IRET
Does simple Interrupt Return, because PIC 8259A is set up for automatic EOI

Figure 3 The ISR of the operating system

The last application intercepting the interval timer interrupt vector 1CH is the first to be serviced, because it saves the ISR address of the previous "first in queue" and puts its own ISR address in place of the one saved.

There is no limit as to the number of applications that can make use of the interrupt.

The overall service time should be kept to a minimum.

6.1.2 Initialization of the interval timer by MS-DOS

This section provides information on how interrupt controller 8259A and timer 2 are initialized by MS-DOS.

The operating system sets up timer 2 for issuing an interrupt every 10 milliseconds (see code listed below). Interrupt type 8 is used.

Timer 2 is set up for square wave generation (Mode 3). This mode does not require a reload of the timer in the Interrupt Service Routine, and it provides the most exact timing. However, Mode 3 requires that interrupt controller 8259A be in edge triggered mode (Bit 3 of ICW1=0). Furthermore, the controller is instructed to handle end of interrupt automatically. Thus, the timer Interrupt Service Routine does not need to reset the interrupt.

Initialization of Interrupt Controller 8259A:

```
CLI                  ; disable Interrupts during Initialization
MOV AL, 00010111B    ; Initialize, edge triggered input, call
                     ; address interval of 4, single, ICW4
OUT 90H, AL          ; PIC 8259A Port address ICW1
MOV AL, 00001000B    ; interrupt type 8 for IRQ 0
OUT 91H, AL          ; PIC 8259A Port address (ICW2)
MOV AL, 00000011B    ; Auto EOI, MCS-86 mode
OUT 91H, AL          ; PIC 8259A Port Address (ICW4)
IN AL, 91H           ; Read Interrupt Mask Register (OCW1)
AND AL, 11111110B    ; Unmask type 8
OUT 91H, AL          ; Write Interrupt mask Register (OCW1)
```

Initialization of Timer 2

```
MOV AL, 10110110B    ; Counter 2, load least significant byte
                     ; then most siginificant byte, Mode 3,
                     ; binary counter (16 bits)
OUT 83H, AL          ; Interval timer 8253 Port (write mode
                     ; word)
MOV AX, 5000 D       ; Valve for counter 2 500 x 500 khz
                       = 10 ms
OUT 82H, AL          ; Interval timer 8253 Port load counter2)
XCHG AL, AH          ; Most sign. byte to AL
OUT 82H, AL
STI                  ; Enable Interrupts
```

6.2 I/O Control Functions

By issuing an I/O control command, you can, among other things, select the cursor and the lines per screen, switch the bell on or off, and tell the system about the installation of a video disk.

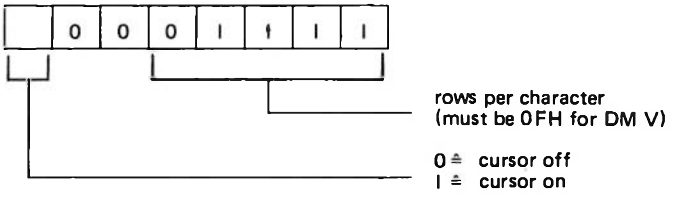
6.2.1 How to write the selected values

A) Select the cursor

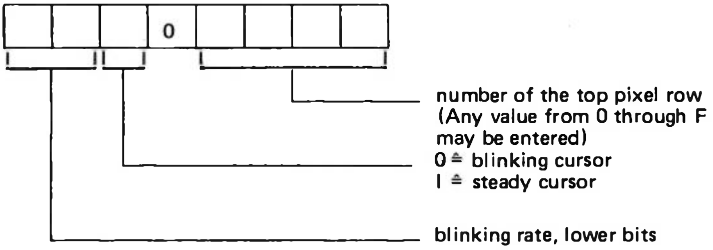
Send the following bytes in an I/O-control-write command to select the cursor:

1. "43H" or "63H" ———————————— Prefix

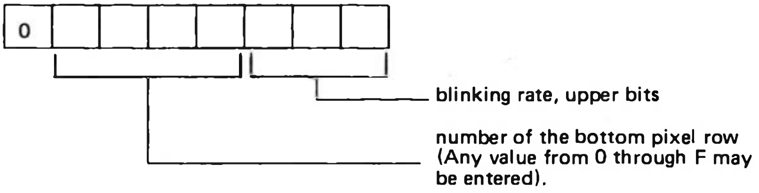
2.


0
0
0
1
1
1
1
rows per character
(must be 0FH for DM V)
0 ≙ cursor off
1 ≙ cursor on

3.


0
number of the top pixel row
(Any value from 0 through F
may be entered)
0 ≙ blinking cursor
1 ≙ steady cursor
blinking rate, lower bits

4.


0
blinking rate, upper bits
number of the bottom pixel row
(Any value from 0 through F may
be entered).

The term "pixel row" may need some brief explanation. The software considers the screen to be made up of pixels. A pixel is simply a dot on the screen. Each of the 25 lines on your screen comprises 16 pixel rows. These pixel rows are numbered 0 through F (from top to bottom).

B) Select the number of lines per screen

Send the following bytes in an I/O-control-write command to select the number of lines on the screen:

1.	"4CH" or "6CH"	Prefix
2.	"18H" or "19H"	hex values for 24 or 25 lines

NOTE: With a screen of 24 lines, the 25th line is reserved for steady displays. Scrolling takes longer.

C) Switch the bell on/off

Send byte "42H" or byte "62H" in an I/O-control-write command to switch the bell on or off. Note that if the bell is switched off, you cannot play music.

D) Installation of a video disk

Send "56H" or "76H" in an I/O-control-write command to tell the system about the installation of a video disk.

6.2.2 How to check the selected values

If an I/O-control-read command is sent, the following eight bytes will be returned:

"43H" or "63H" ——— Prefix

☐
☐ ——— cursor type (the 3 bytes defining the cursor; compare with A))
☐

"4CH" or "6CH" ——— Prefix

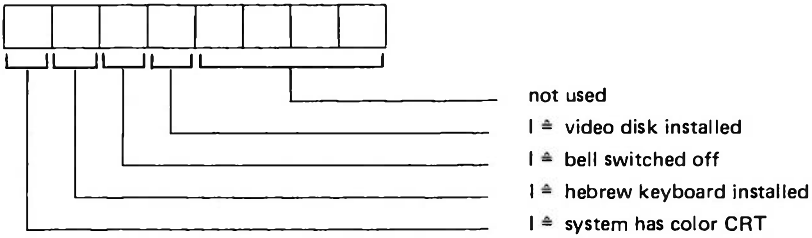
☐ ——— number of lines (1 byte, either 18H or 19H; compare with B))

"42H" or "62H" Prefix

☐ ——— console flags (1 byte, compare with the next paragraph, "Pattern of the console 'console flags' byte".)

6.2.3 The pattern of the "console flags" byte

The byte "console flags" has the following pattern:


not used
1 ≙ video disk installed
1 ≙ bell switched off
1 ≙ hebrew keyboard installed
1 ≙ system has color CRT

```
;   US-ENGLISH

;F1 !F2 !F3 !F4 !F5 !F6 !F7 !F8 !F9 !F10!F11!F12!F13!F14!F15!
; A0! A1! A2! A3! A4! A5! A6! A7! A8! A9! AA! AB! AC! AD! AE!
; C0! C1! C2! C3! C4! C5! C6! C7! C8! C9! CA! CB! CC! CD! CE!
; E0! E1! E2! E3! E4! E5! E6! E7! E8! E9! EA! EB! EC! ED! EE!
;---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
;ESC!1! !2@ !3# !4$ !5% !6^ !7& !8* !9( !0) !-_ !=+ !BSF!TAB!
; 1B!   !   !   !   !   !   !   !   !   !   ! 1F!   ! 9B! 09!
; 1B! 21! 40! 23! 24! 25! 5E! 26! 2A! 28! 29! 5F! 2B! 8B! 09!
; 1B! 31! 32! 33! 34! 35! 36! 37! 38! 39! 30! 2D! 3D! 8B! 09!
;------------------------------------------------------------
;     !qQ !wW !eE !rR !Tt !yY !uU !iI !oO !pP ![{ !]} !     !
;     ! 11! 17! 05! 12! 14! 19! 15! 09! 0F! 10! 1B! 1D!     !
;CONTL! 51! 57! 45! 52! 54! 59! 55! 49! 4F! 50! 7B! 7D!CONTL!
;     ! 71! 77! 65! 72! 74! 79! 75! 69! 6F! 70! 5B! 5D!     !
;------------------------------------------------------------
;!     !aA !sS !dD !fF !gG !hH !jJ !kK !lL !;: !'" !`~ !   !
;!     ! 01! 13! 04! 06! 07! 08! 0A! 0B! 0C!   ! 00! 1E! 98!
;!LOCK^! 41! 53! 44! 46! 47! 48! 4A! 4B! 4C! 3A! 22! 7E! 88!
;!     ! 61! 73! 64! 66! 67! 68! 6A! 6B! 6C! 3B! 27! 60! 88!
;------------------------------------------------------- FT!
;!   !\| !zZ !xX !cC !vV !bB !nN !mM !,< !.> !/? !     !   !
;! ^ ! 1C! 1A! 18! 03! 16! 02! 0E! 0D!   !   !   !  ^  !   !
;!SHF! 7C! 5A! 58! 43! 56! 42! 4E! 4D! 3C! 3E! 3F!SHIFT!   !
;!   ! 5C! 7A! 78! 63! 76! 62! 6E! 6D! 2C! 2E! 2F!     !   !
;!----------------------------------------------------------
;            !                                !
;            !               20               !
;            !               20               !
;            !               20               !
;            ----------------------------------


;   UK-ENGLISH

;F1 !F2 !F3 !F4 !F5 !F6 !F7 !F8 !F9 !F10!F11!F12!F13!F14!F15!
; A0! A1! A2! A3! A4! A5! A6! A7! A8! A9! AA! AB! AC! AD! AE!
; C0! C1! C2! C3! C4! C5! C6! C7! C8! C9! CA! CB! CC! CD! CE!
; E0! E1! E2! E3! E4! E5! E6! E7! E8! E9! EA! EB! EC! ED! EE!
;---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
;ESC!1! !2" !3£ !4$ !5% !6& !7' !8( !9) !0_ !-= !^~ !BSF!TAB!
; 1B!   !   !   !   !   !   !   !   !   ! 1F!   ! 1E! 9B! 09!
; 1B! 21! 22! 23! 24! 25! 26! 27! 28! 29! 5F! 3D! 7E! 8B! 09!
; 1B! 31! 32! 33! 34! 35! 36! 37! 38! 39! 30! 2D! 5E! 8B! 09!
;------------------------------------------------------------
;     !qQ !wW !eE !rR !tT !yY !uU !iI !oO !pP !@` ![{ !     !
;     ! 11! 17! 05! 12! 14! 19! 15! 09! 0F! 10! 00! 1B!     !
;CONTL! 51! 57! 45! 52! 54! 59! 55! 49! 4F! 50! 60! 7B!CONTL!
;     ! 71! 77! 65! 72! 74! 79! 75! 69! 6F! 70! 40! 5B!     !
;------------------------------------------------------------
;!     !aA !sS !dD !fF !gG !hH !jJ !kK !lL !;+ !:* !]} !   !
;!     ! 01! 13! 04! 06! 07! 08! 0A! 0B! 0C!   !   ! 1D! 98!
;!LOCK^! 41! 53! 44! 46! 47! 48! 4A! 4B! 4C! 2B! 2A! 7D! 88!
;!     ! 61! 73! 64! 66! 67! 68! 6A! 6B! 6C! 3B! 3A! 5D! 88!
;------------------------------------------------------- FT!
;!   !\| !zZ !xX !cC !vV !bB !nN !mM !,< !.> !/? !     !   !
;! ^ ! 1C! 1A! 18! 03! 16! 02! 0E! 0D!   !   !   !  ^  !   !
;!SHF! 7C! 5A! 58! 43! 56! 42! 4E! 4D! 3C! 3E! 3F!SHIFT!   !
;!   ! 7C! 7A! 78! 63! 76! 62! 6E! 6D! 2C! 2E! 2F!     !   !
;!----------------------------------------------------------
;            !               20               !
;            !               20               !
;            !               20               !
;            ----------------------------------
```

```
;NUMERIC KEY PAD PART
;      !F16!F17!F18!F19!F20!
;      ! AF! B0! B1! B2! B3!
;      ! CF! D0! D1! D2! D3!
;      ! EF! F0! F1! F2! F3!
;      !---+---+---+---+---!
;      ↖   !←  ! ↓ : ↑ ! → !
;      ! 91! 92! 93! 94! 95!
;      ! 81! 82! 83! 84! 85!
;      ! 81! 82! 83! 84! 85!
;      !---+---+---+---+---!
;      !CLR!7  !8  !9  !/  !
;      ! 96! B7! B8! B9! BF!
;      ! 7F! D7! D8! D9! DF!
;      ! 86! 37! 38! 39! 2F!
;      !---+---+---+---+---!
;      !-  !4  !5  !6  !*  !
;      ! F5! B4! B5! B6! BA!
;      ! F4! D4! D5! D6! DA!
;      ! 2D! 34! 35! 36! 2A!
;      !---+---+---+---+---!
;      !+  !1  !2  !3  !FT !
;      ! BB! BC! BD! BE! 98!
;      ! DB! DC! DD! DE! 88!
;      ! 2B! 31! 32! 33! 88!
;      !-------+---+---!   !
;      !0      !00 !.  !   !
;      ! 30    !*30!   !<--------used to switch fm . to , or back
;      ! 30    !*30! 2E!<--------OUTPUT: 2EH(.) or 2CH(,) as set
;      ! 30    !*30! 2E!   !
;      ---------------------
```

EXPLANATION: 1ST LINE SYMBOLS, 2ND LINE CONROL-CODE,
3RD LINE SHIFT-CODE, 4TH UNSHIFT

For some keyboards, the keys listed below are designed as follows:

F1
A0 — —volume control for the bell (—)
C0
E0

F2
A1 — —volume control for the bell (+)
C1
E1

F3
A2 — —brightness (—)
C2
E2

F4
A3 — —brightness (+)
C3
E3

F20
B3 — —used for system reset
D3
F3

.
— — — —used for switching from point to comma and back
2E OUTPUT:2EH (.) or 2CH (,); older keyboards return
2E 9AH in CONTROL mode, and 8AH instead of 2EH and 2CH.

INDEX

.COM file 2-12

Absolute Disk Read (Interrupt 25H) 1-23
Absolute Disk Write (Interrupt 26H) 1-25
Allocate Memory (Function 48H) 1-128
Archive bit 3-6
ASCIZ 1-107
Attribute field 2-4
Attributes 1-12
AUTOEXEC file 3-2
Auxiliary Input (Function 03H) 1-36
Auxiliary Output (Function 04H) 1-37

Basic 1-1
BIOS 1-25, 2-6
BIOS Parameter Block 2-10, 2-13
Bit 8 2-9
Bit 9 2-9
Block devices 2-1, 2-8, 2-10
Boot sector 2-14
BPB 2-10
BPB pointer 2-12
Buffered Keyboard Input (Function 0AH) 1-45
BUILD BPB 2-4, 2-8, 2-13
Busy bit . . . . 2-9, 2-17 to 2-18

Case mapping 1-108
Change Attributes (Function 43H) 1-120
Change Current Directory (Function 3BH) 1-111
Character device 2-1, 2-5
  Example 2-34
Check Keyboard Status (Function 0BH) 1-47
CLOCK device 2-4, 2-19
Close a File Handle (Function 3EH) 1-115
Close File (Function 10H) 1-53
Cluster 3-3

Command code field 2-7
Command processor 3-2
COMMAND.COM . . . . 3-1 to 3-2
COMSPEC= 4-3
CON device 2-5
CONFIG.SYS 2-6, 2-12
Console input/output calls 1-3
Control blocks 4-1
Control information 5-1
CONTROL-C Check (Function 33H) 1-102
CONTROL-C Exit Address (Interrupt 23H) 1-19, 3-2
CP/M-compatible calling sequence 1-28
Create a File (Function 3CH) 1-112
Create File (Function 16H) 1-65
Create Sub-Directory (Function 39H) 1-109
Current Disk (Function 19H) 1-69

DATE 2-19
Delete a Directory Entry (Function 41H) 1-118
Delete File (Function 13H) 1-59
Device drivers 3-7
  Creating 2-5
  Dumb 2-11

  Installing 2-6
  Smart 2-11
Device header 2-3
Direct Console I/O (Function 06H) 1-40
Direct Console Input (Function 07H) 1-42
Directory entry 1-6
Disk allocation 3-3
Disk Directory 3-4
Disk errors 1-22
Disk format
  IBM 3-3
  MS-DOS 3-7
Disk I/O calls 1-3
Disk Reset (Function 0DH) 1-49
Disk Transfer Address 1-63, 4-3
Display Character (Function 02H) 1-35
Display String (Function 09H) 1-44
Done bit 2-9
Driver 2-2
Dumb device driver 2-11

Duplicate a File Handle (Function 45H) 1-125

Error codes 1-20
EXE files 5-1
Extended File Control Block 1-6

FAT 1-11, 2-8, 2-13, 3-3, 3-7
FAT ID byte 2-13, 2-15
Fatal Error Abort Address (Interrupt 24H) 1-20, 3-2
FCB 1-3
File Allocation Table 1-11, 3-3, 3-7
File Control Block 1-3, 1-51
  Extended 1-6, 4-10
  Fields 1-4, 1-7
  Opened 1-3
  Unopened 1-3
File control Block 1-3
File Size (Function 23H) 1-76
Filename separators 1-88
Filename terminators 1-88
Find Match File (Function 4EH) 1-136
FLUSH 2-18
Flush Buffer (Function 0CH) 1-48
Force Duplicate of Handle (Function 46H) 1-126
FORMAT 3-4
Fortran 1-2
Free Allocated Memory (Function 49H) 1-129
Function call parameters 2-11
Function dispatcher 1-28
Function Request (Interrupt 21H) 1-18, 4-3
Function Requests
  Function 00H 1-33
  Function 01H 1-34
  Function 02H 1-35
  Function 03H 1-36
  Function 04H 1-37
  Function 05H 1-38
  Function 06H 1-40
  Function 07H 1-42
  Function 08H 1-43

Function 09H 1-44
Function 0AH 1-45
Function 0BH 1-47
Function 0CH 1-48
Function 0DH 1-49, 1-63
Function 0EH 1-50
Function 0FH 1-51, 1-65
Function 10H 1-53
Function 11H 1-55
Function 12H 1-57
Function 13H 1-59
Function 14H 1-61
Function 15H 1-63
Function 16H 1-65
Function 17H 1-67
Function 19H 1-69
Function 1AH 1-70
Function 21H 1-72
Function 22H 1-74
Function 23H 1-76
Function 24H 1-78
Function 25H 1-19, 1-79
Function 27H 1-81
Function 28H 1-84
Function 29H 1-87
Function 2AH 1-90
Function 2BH 1-92
Function 2CH 1-94
Function 2DH 1-95
Function 2EH 1-97
Function 2FH 1-99
Function 30H 1-100
Function 31H 1-101
Function 33H 1-102
Function 35H 1-104
Function 36H 1-105
Function 38H 1-106
Function 39H 1-109
Function 3AH 1-110
Function 3BH 1-111
Function 3CH 1-112
Function 3DH 1-113
Function 3EH 1-115
Function 3FH 1-116

Function 40H 1-117
Function 41H 1-118
Function 42H 1-119
Function 43H 1-120
Function 44H 1-121
Function 45H 1-125
Function 46H 1-126
Function 47H 1-127
Function 48H 1-128
Function 49H 1-129
Function 4AH 1-130
Function 4BH 1-131
Function 4CH 1-134
Function 4DH 1-135
Function 4EH 1-136
Function 4FH 1-138
Function 54H 1-139
Function 56H 1-140
Function 57H 1-141
Function OAH 1-45

Get Date (Function 2AH) 1-90
Get Disk Free Space (Function 36H) 1-105
Get Disk Transfer Address (Function 2FH) 1-99
Get DOS Version Number (Function 30H) 1-100
Get Interrupt Vector (Function 35H) 1-104
Get Time (Function 2CH) 1-94
Get/Set Date/Time of File (Function 57H) 1-141

Header 5-1
Hidden files 1-57, 3-5
Hierarchical directories 1-11
High-level languages 1-1

I/O Control for Devices (Function 44H) 1-121, 2-4
I/O Control Functions 6-6
IBM disk format 3-3
INIT . . . . 2-5, 2-10 to 2-12
Initial allocation block 1-101
Installable device drivers 2-5
Instruction Pointer 4-4
Internal stack 1-29
Interrupt entry point 2-1
Interrupt handlers 1-19, 4-1
Interrupt-handling routine 1-80

Interrupts 1-14
  Interrupt 16H 1-16
  Interrupt 20H 1-17, 1-33
  Interrupt 21 H 1-18, 1-28
  Interrupt 22 H 1-19
  Interrupt 23H . . . . 1-19, 1-34 to 1-35, 1-38, 1-43, 1-45
  Interrupt 24H 1-20
  Interrupt 25H 1-23
  Interrupt 26H 1-25
  Interrupt 27H 1-27
IO.SYS 3-1, 3-6
IOCIL bit 2-4

Keep Process Function 31H) 1-101
Keyboard Character Code Read (Interrupt 16H) 1-16
Keyboard Code Charts A-1

Load and Execute Program (Function 4BH) 1-131
Load module . . . . 5-1 to 5-2
Local buffering 2-6
Logical sector 3-7
Logical sector number 3-8

Macro 1-10
MEDIA CHECK 2-8, 2-12
Media descriptor byte . . . . 2-10 to 2-11, 2-15
Modify Allocated Memory Blocks (Function 4AH) 1-130
Move a Directory Entry (Function 56H) 1-140
Move File Pointer (Function 42H) 1-119
MS-DOS initialization 3-1
MS-DOS memory map 4-1
MS-LINK . . . . 5-1 to 5-2
MSDOS.SYS . . . . 3-1 to 3-2, 3-6
Multiple media 2-11

Name field 2-5
NON DESTRUCTIVE READ NO WAIT 2-17
Non IBM format 2-8
Non IBM format bit 2-4, 2-13
NUL device 2-4

Offset 50H 1-28
Open a File (Function 3DH) 1-113
Open File (Function 0FH) 1-51

Parse File Name (Function 29H) 1-87
Pascal 1-2

PATH 4-3
Pattern of the "Console Flags" Byte 6-8
Pointer to Next Device field 2-3
Print Character (Function 05H) 1-38
Printer input/output calls 1-3
Program segment 4-2
Program Segment Prefix . . . . . 1-2 to 1-3, 1-20, 1-28, 4-2
Program Terminate (Interrupt 20H) 1-17
PROMPT 4-3

Random Block Read (Function 27H) 1-81
Random Block Write (Function 28H) 1-84
Random Read (Function 21H) 1-72
Random Write (Function 22H) 1-74
Read From File/Device (Function 3FH) 1-116
Read Keyboard (Function 08H) 1-43
Read Keyboard and Echo (Function 01H) 1-34
Read Only Memory 3-1
READ or WRITE 2-16
Record Size 1-63
Registers 1-29
Relocation information 5-1
Relocation item offset value 5-3
Relocation table 5-2
Remove a Directory Entry (Function 3AH) 1-110
Rename File (Function 17H) 1-67
Request Header 2-6
Retrieve Return Code (Function 4DH) 1-135
Return Country-Dependent Info (Function 38H) 1-106
Return Current Setting (Function 54H) 1-139
Return Text of Current Directory (Function 47H) 1-127
Returning control to MS-DOS 1-2
ROM 3-1
Root directory 1-11, 3-4

Search for First Entry (Function 11H) 1-55
Search for Next Entry (Function 12H) 1-57
Select Disk (Function 0EH) 1-50
Sequential Read (Function 14H) 1-61
Sequential Write (Function 15H) 1-63
SET 4-3
Set Date (Function 2BH) 1-92
Set Disk Transfer Address (Function 1AH) 1-70
Set Relative Record (Function 24H) 1-78
Set Time (Function 2DH) 1-95

Set Vector (Function 25H) 1-19, 1-79
Set/Reset Verify Flag (Function 2EH) 1-97
Smart device driver 2-11
Start segment value 5-3
STATUS 2-18
Status word 2-9
Step Through Directory (Function 4FH) 1-138
Strategy entry point 2-1
Strategy routines 2-5
System files 1-57, 3-5
System prompt 3-2

Terminate a Process (Function 4CH) 1-134
Terminate Address (Function 4CH) 4-2
Terminate Address (Interrupt 22H) 1-19, 3-2
Terminate But Stay Resident (Interrupt 27H) 1-27
Terminate Program (Function 00H) 1-33
TIME 2-19
Timer Interrupt Support 6-1
Type-ahead buffer 2-18

Unit code 2-7
User stack 1-21, 4-1

Volume label 3-5

Wild card characters 1-57, 1-59, 1-88
Write to a File/Device (Function 40H) 1-117

Xenix-compatible calls 1-11

MS-LIB Library Manager

MS-LIB
CONTENTS

Introduction
Features and Benefits of MS-LIB
Overview of MS-LIB Operation 4

Chapter 1 RUNNING MS-LIB

1.1 Invoking MS-LIB 1-1
1.1.1 Method 1: LIB 1-2
Summary of Command Prompts 1-2
Summary of Command Characters 1-2
1.1.2 Method 2: LIB <library> <operations>, <listing> 1-3
1.1.3 Method 3: LIB @ <filespec> 1-5
1.2 Command Prompts 1-7
1.3 Command Characters 1-9
+ - append 1-9
— - delete 1-9
* - extract 1-10
; - default remaining prompts 1-10
& - continuation 1-11
Control-C - program abort 1-11

Chapter 2 ERROR MESSAGES

INTRODUCTION

Features and Benefits

MS-LIB creates and modifies library files that are used with Microsoft's MS-LINK Linker Utility. MS-LIB can add object files to a library, delete modules from a library, or extract modules from a library and place the extracted modules into separate object files.

MS-LIB provides a means of creating either general or special libraries for a variety of programs or for specific programs only. With MS-LIB you can create a library for a language compiler, or you can create a library for one program only, which would permit very fast linking and possibly more efficient execution.

You can modify individual modules within a library by extracting the modules, making changes, then adding the modules to the library again. You can also replace an existing module with a different module or with a new version of an existing module.

The command scanner in MS-LIB is the same as the one used in Microsoft's MS-LINK, MS-Pascal, MS-FORTRAN, and other 16-bit Microsoft products. If you have used any of these products, using MS-LIB is familiar to you. Command syntax is straightforward, and MS-LIB prompts you for any of the commands it needs that you have not supplied. There are no surprises in the user interface.

Overview of MS-LIB Operation

MS-LIB performs two basic actions: it deletes modules from a library file, and it changes object files into modules and appends them to a library file. These two actions underlie five library manager functions:

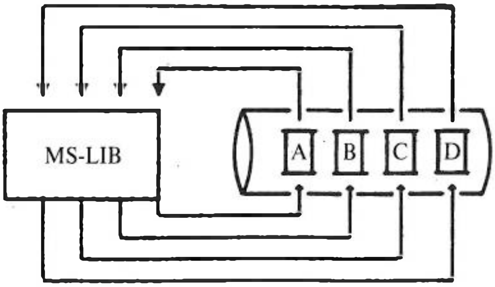
delete a module
extract a module and place it in a separate object file
append an object file as a module of a library
replace a module in the library file with a new module
create a library file

During each library session, MS-LIB first deletes or extracts modules, then appends new ones. In a single operation, MS-LIB reads each module into memory, checks it for consistency, and writes it back to the file. If you delete a module, MS-LIB reads in that module but does not write it back to the file. When MS-LIB writes back the next module to be retained, it places the module at the end of the last module written. This procedure effectively "closes up" the disk space to keep the library file from growing larger than necessary. When MS-LIB has read through the whole library file, it appends any new modules to the end of the file. Finally, MS-LIB creates the index, which MS-LINK uses to find modules and symbols in the library file, and outputs a cross reference listing of the PUBLIC symbols in the library, if you request such a listing. (Building the library index may take some extra time, up to 20 seconds in some cases.)

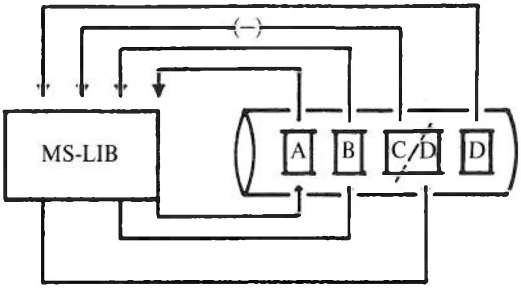
For example:

```
LIB PASCAL+HEAP-HEAP;
```

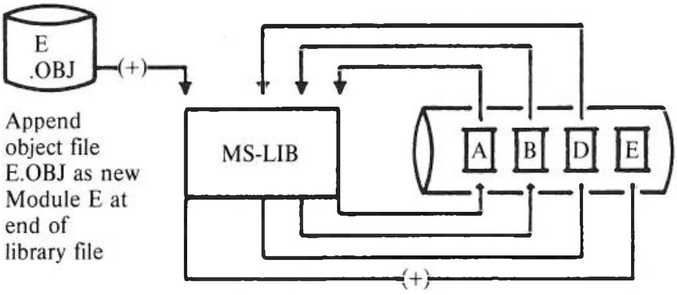
first deletes the library module HEAP from the library file, then adds the file HEAP.OBJ as the last module in the library. This order of execution prevents confusion in MS-LIB when a new version of a module replaces a version in the library file. Note that the replace function is simply the delete-append functions in succession. Also note that you can specify delete, append, or extract functions in any order; the order is insignificant to the MS-LIB command scanner.



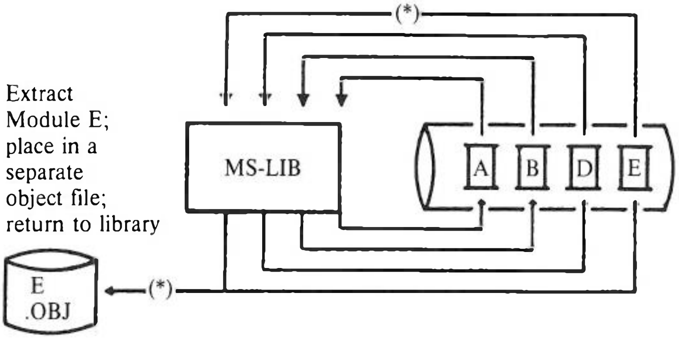
Consistency
Check only


(−)
MS-LIB
A
B
C/D
D

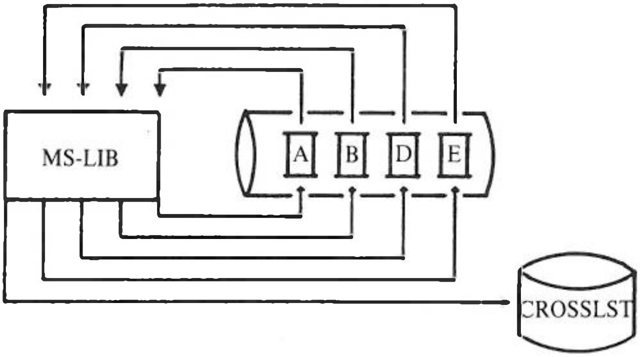
Delete
Module C;
Module D
written to
space of
Module C


E
.OBJ
(+)
MS-LIB
A
B
D
E
(+)

Append
object file
E.OBJ as new
Module E at
end of
library file


(*)
MS-LIB
A
B
D
E
E
.OBJ
(*)

Extract Module E; place in a separate object file; return to library


MS-LIB
A
B
D
E
CROSSLST

Consistency Check, then output a cross reference listing of PUBLIC symbols

CHAPTER 1
RUNNING MS-LIB

Running MS-LIB requires two types of commands: a command to invoke MS-LIB and answers to command prompts. Usually you will enter all the commands to MS-LIB on the terminal keyboard. As an option, answers to the command prompts may be contained in a Response File. Some special command characters exist. Some are used as a required part of MS-LIB commands. Others assist you while entering MS-LIB commands.

1.1 INVOKING MS-LIB

MS-LIB may be invoked three ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the line used to invoke MS-LIB. By the third method, you create a Response File that contains all the necessary commands.

Summary of Methods to invoke MS-LIB

Method 1	LIB
Method 2	LIB <library> <operations>,<listing>
Method 3	LIB @ <filespec>

1.1.1 Method 1: LIB

Enter:

```
LIB
```

MS-LIB will be loaded into memory. Then, MS-LIB returns a series of three text prompts that appear one at a time. You answer the prompts as commands to MS-LIB to perform specific tasks.

The Command Prompts and Command Characters are summarized here. The Command Prompts and Command Characters are described fully in Sections 1.2 and 1.3.

Summary of Command Prompts

PROMPT	RESPONSES
Library file:	List filename of library to be manipulated (default: filename extension .LIB)
Operation:	List command character(s) followed by module name(s) or object filename(s) (default action: no changes. default object filename extension: .OBJ)
List file:	List filename for a cross reference listing file (default: NUL; no file)

Summary of Command Characters

Character	Action
+	Append an object file as the last module
-	Delete a module from the library
*	Extract a module and place in an object file
;	Use default responses to remaining prompts
&	Extend current physical line; repeat command prompt
Control-C	Abort library session.

1.1.2 Method 2: LIB <library> <operations>,<listing>

Enter:

LIB <library> <operations>,<listing>

The entries following LIB are responses to the command prompts. The **library** and **operations** fields and all operations entries must be separated by one of the command characters plus, minus, and asterisk (+, -, *). If a cross reference listing is wanted, the name of the file must be separated from the last operations entry by a comma.

where: **library** is the name of a library file. MS-LIB assumes that the filename extension is .OBJ, which you may override by specifying a different extension. If the filename given for the **library** fields does not exist, MS-LIB will prompt you:

Library file does not exist. Create?

Enter Yes (or any response beginning with Y) to create a new library file. Enter No (or any other response not beginning with Y) to abort the library session.

operations is deleting a module, appending an object file as a module, or extracting a module as an object file from the library file. Use the three command characters plus (+), minus (-), and asterisk (*) to direct MS-LIB what to do with each module or object file.

listing is the name of the file you want to receive the cross reference listing of PUBLIC symbols in the modules in the library. The list is compiled after all module manipulation has taken place.

To select the default for remaining field(s), you may enter the semicolon command character.

If you enter a Library filename followed immediately by a semicolon, MS-LIB will read through the library file and perform a consistency check. No changes will be made to the modules in the library file.

If you enter a Library filename followed immediately by a comma and a List filename, MS-LIB will perform its consistency check of the library file, then produce the cross reference listing file.

Example

```
LIB PASCAL-HEAP+HEAP;
```

This example causes MS-LIB to delete the module HEAP from the library file PASCAL.LIB, then append the object file HEAP.OBJ as the last module of PASCAL.LIB (the module will be named HEAP).

If you have many operations to perform during a library session, use the ampersand (&) command character to extend the line so that you can enter additional object filenames and module names. Be sure to always include one of the command characters for operations (+, -, *) before the name of each module or object filename.

Example

```
LIB PASCAL<CR>
```

causes MS-LIB to perform a consistency check of the library file PASCAL.LIB. No other action is performed.

Example

```
LIB PASCAL,PASCROSS.PUB
```

causes MS-LIB to perform a consitency check of the library file PASCAL.LIB, then output a cross reference listing file named PASCROSS.PUB.

1.1.3 Method 3: LIB @ <filespec>

Enter:

LIB @ <filespec>

where: **filespec** is the name of a Response File. A Response File contains answers to the MS-LIB prompts (summarized under method 1 for invoking and described fully in Section 1.2). Method 3 permits you to conduct the MS-LIB session without interactive (direct) user responses to the MS-LIB prompts.

IMPORTANT

Before using method 3 to invoke MS-LIB, you must first create the Response File.

A Response File has text lines, one for each prompt. Responses must appear in the same order as the command prompts appear.

Use Command Characters in the Response File the same way as they are used for responses entered on the terminal keyboard.

When the library session begins, each prompt will be displayed in turn with the responses from the response file. If the response file does not contain answers for all the prompts, MS-LIB will use the default responses (no changes to the modules currently in the library file for Operation, and no cross reference listing file created).

If you enter a Library filename followed immediately by a semicolon, MS-LIB will read through the library file and perform a consistency check. No changes will be made to the modules in the library file.

If you enter a Library filename then only a carriage return of Operations then a comma and a List filename, MS-LIB will perform its consistency check of the library file, then produce the cross reference listing file.

Example:

```
PASCAL<CR>
+CURSOR+HEAP-HEAP*FOIBLES<CR>
CROSSLST<CR>
```

This Response File will cause MS-LIB to delete the module HEAP from the PASCAL.LIB library file, extract the module FOIBLES and place in an object file named FOIBLES.OBJ, then append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Then, MS-LIB will create a cross reference file named CROSSLST.

1.2 COMMAND PROMPTS

MS-LIB is commanded by entering responses to three text prompts. When you have entered your response to the current prompt, the next appears. When the last prompt has been answered, MS-LIB performs its library management functions without further command. When the library session is finished, MS-LIB exits to the operating system. When the operating system prompt is displayed, MS-LIB has finished the library session successfully. If the library session is unsuccessful, MS-LIB returns the appropriate error message.

MS-LIB prompts you for the name of the library file, the operation(s) you want to perform, and the name you want to give to a cross reference listing file, if any.

Library file:

Enter the name of the library file that you want to manipulate. MS-LIB assumes that the filename extension is .LIB. You can override this assumption by giving a filename extension when you enter the library filename. Because MS-LIB can manage only one library file at a time, only one filename is allowed in response to this prompt. Additional responses, except the semicolon command character, are ignored.

If you enter a library filename and follow it immediately with a semicolon command character, MS-LIB will perform a consistency check only, then return to the operating system. Any errors in the file will be reported.

If the filename you enter does not exist, MS-LIB returns the prompt:

```
Library file does not exist. Create?
```

You must enter either Yes or No, in either upper or lower (or mixed) case. Actually, MS-LIB checks the response of the letter Y as the first character. If any other character is entered first, MS-LIB terminates and returns to the operating system.

Operation:

Enter one of the three command characters for manipulating modules (+, –, *), followed immediately (no space) by the module name or the object filename. Plus sign appends an object file as the last module in the library file (see further discussion under the description of plus sign below). Minus sign deletes a module from the library file. Asterisk extracts a module from the library and places it in a separate object file with the filename taken from the module name and a filename extension .OBJ.

When you have a large number of modules to manipulate (more than can be typed on one line), enter an ampersand (&) as the last character on the line. MS-LIB will repeat the Operation prompt, which permits you to enter additional module names and object filenames.

MS-LIB allows you to enter operations on modules and object files in any order you want.

More information about order of execution and what MS-LIB does with each module is given in the descriptions of each Command Character.

List file:

If you want a cross reference list of the PUBLIC symbols in the modules in the library file after your manipulations, enter a filename in which you want MS-LIB to place the cross reference listing. If you do not enter a filename, no cross reference listing is generated (a NUL file).

The response to the List file prompt is a file specification. Therefore, you can specify, along with the filename, a drive (or device) designation and a filename extension. The List file is not given a default filename extension. If you want the file to have a filename extension, you must specify it when entering the filename.

The cross reference listing file contains two lists. The first list is an alphabetical listing of all PUBLIC symbols. Each symbol name is followed by the name of its module. The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the PUBLIC symbols in that module.

1.3 COMMAND CHARACTERS

MS-LIB provides six command characters: three of the command characters are required in responses to the Operation prompt; the other three command characters provide you additional helpful commands to MS-LIB.

+ The plus sign followed by an object filename appends the object file as the last module in the library named in response to the Library file prompt. When MS-LIB sees the plus sign, it assumes that the filename extension is .OBJ. You may override this assumption by specifying a different filename extension.

MS-LIB strips the drive designation and the extension from the object file specification, leaving only the filename. For example, if the object file to be appended as a module to a library is:

```
B:CURSOR.OBJ
```

a response to the Operation prompt of:

```
+B:CURSOR.OBJ
```

causes MS-LIB to strip off the B: and the .OBJ, leaving only CURSOR, which becomes a module named CURSOR in the library.

NOTE

The distinction between an object file and a module (or object module) is that the file possesses a drive designation (even if it is default drive) and a filename extension. Object modules possess neither of these.

- The minus sign followed by a module name deletes that module from the library file. MS-LIB then "closes up" the file space left empty by the deletion. This cleanup action keeps the library file from growing larger than necessary with empty space. Remember that new modules, even replacement modules are added to the end of the file, not stuffed into space vacated by deleting modules.

*

The asterisk followed by a module name extracts that module from the library file and places it into a separate object file. The module will still exist in the library (extract means, essentially, copy the module to a separate object file). The module name is used as the filename. MS-LIB adds the default drive designation and the filename extension .OBJ. For example, if the module to be extracted is:

```
CURSOR
```

and the current default disk drive is A:, a reponse to the Operation prompt of:

```
*CURSOR
```

causes MS-LIB to extract the module named CURSOR from the library file and to set it up as an object file with the file specification of:

```
default drive:CURSOR.OBJ
```

(The drive designation and filename extension cannot be overridden. You can, however, rename the file, giving a new filename extension, and/or copy the file to a new disk drive, giving a new filename and/or filename extension.)

;

Use a single semicolon (;) followed immediately by a carriage return at any time after responding to the first prompt (from Library file on) to select default responses to the remaining prompts. This feature saves time and overrides the need to answer additional prompts.

NOTE

Once the semicolon has been entered, you can no longer respond to any of the prompts for that library session. Therefore, do not use the semicolon to skip over some prompts. For this, use carriage return.

Example:

```
Library file: FUN <CR>
Operation: +CURSOR;<CR>
```

The remaining prompt will not appear, and MS-LIB will use the default value (no cross reference file).

& Use the ampersand to extend the current physical line. This command character will only be needed for the Operation prompt. MS-LIB can perform many functions during a single library session. The number of modules you can append is limited only by disk space. The number of modules you can replace or extract is also limited only by disk space. The number of modules you can delete is limited only by the number of modules in the library file. However, the line length for a response to any prompt is limited to the line length of your system. For a large number of responses to the Operation prompt, place an ampersand at the end of a line. MS-LIB will display the Operation prompt again, then enter more responses. You may use the ampersand character as many times as you need. For example:

```
Library file: FUN<NEW LINE>
Operation: +CURSOR-HEAP+HEAP*FOIBLES&
Operation: *INIT+ASSUME+RIDE;<NEW LINE>
```

MS-LIB will delete the module HEAP, extract the modules FOIBLES and INIT (creating two files, FOIBLES.OBJ and INIT.OBJ), then append the object files CURSOR, HEAP, ASSUME, and RIDE. Note, however, that MS-LIB allows you to enter your Operation responses in any order.

Control-C

Use Control-C at any time to abort the library session. If you enter an erroneous response, such as the wrong filename or module name, or an incorrectly spelled filename or module name, you must press CTRL-C to exit MS-LIB then reinvoke MS-LIB and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

CHAPTER 2
ERROR MESSAGES

<symbol> is a multiply defined PUBLIC. Proceed?

Cause: two modules define the same public symbol. The user is asked to confirm the removal of the definition of the old symbol. A No response leaves the library in an undetermined state.

Cure: Remove the PUBLIC declaration from one of the object modules and recompile or reassemble.

Allocate error on VM.TMP
  Cause: out of space
Cannot create extract file
  Cause: no room in directory for extract file
Cannot create list file
  Cause: No room in directory for library file
Cannot nest response file
  Cause: "@filespec" in response (or indirect) file
Cannot open VM.TMP
  Cause: no room for VM.TMP in disk directory
Cannot write library file
  Cause: Out of space
Close error on extract file
  Cause: out of space
Error: An internal error has occurred.
  Contact Microsoft, Inc.
Fatal Error: Cannot open input file
  Cause: Mistyped object file name
Fatal Error: Module is not in the library
  Cause: trying to delete a module that is not in the library
Input file read error
  Cause: bad object module or faulty disk
Invalid object module/library
  Cause: bad object and/or library
Library Disk is full
  Cause: no more room on diskette
Listing file write error
  Cause: out of space

No library file specified
    Cause: no response to Library File prompt
Read error on VM.TMP
    Cause: disk not ready for read
Symbol table capacity exceeded
    Cause: too many public symbols (about 30K chars in symbols)
Too many object modules
    Cause: more than 500 object modules
Too many public symbols
    Cause: 1024 public symbols maximum
Write error on library/extract file
    Cause: Out of space
Write error on VM.TMP
    Cause: out of space

NCR

DEBUG Utility

DEBUG UTILITY
CONTENTS

Chapter 1 INTRODUCTION

1.1 Overview of DEBUG 1-1
1.2 How to Start DEBUG 1-1

Chapter 2 COMMANDS

2.1 Command Information 2-1
2.2 Parameters 2-3
2.3 Error Messages 2-36

CHAPTER 1
INTRODUCTION

1.1 OVERVIEW OF DEBUG

The Microsoft DEBUG Utility (DEBUG) is a debugging program that provides a controlled testing environment for binary and executable object files. Note that EDLIN is used to alter source files; DEBUG is EDLIN's counterpart for binary files. DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change. It allows you to alter the contents of a file or the contents of a CPU register, and then to immediately reexecute a program to check on the validity of the changes.

All DEBUG commands may be aborted at any time by pressing <CONTROL-C>. <CONTROL-S> suspends the display, so that you can read it before the output scrolls away. Entering any key other than <CONTROL-C> or <CONTROL-S> restarts the display. All of these commands are consistent with the control character functions available at the MS-DOS command level.

1.2 HOW TO START DEBUG

DEBUG may be started in two ways. By the first method, you type all commands in response to the DEBUG prompt (a hyphen). By the second method, you type all commands on the line used to start DEBUG.

Summary of Methods to Start DEBUG

Method 1	DEBUG
Method 2	DEBUG [<filespec> [<arglist>]]

1.2.1 Method 1: DEBUG

To start DEBUG using method 1, type:

```
DEBUG
```

DEBUG responds with the hyphen (-) prompt, signaling that it is ready to accept your commands. Since no filename has been specified, current memory, disk sectors, or disk files can be worked on by using other commands.

Warnings

1. When DEBUG is started, it sets up a program header at offset 0 in the program work area. On previous versions of DEBUG, you could overwrite this header. You can still overwrite the default header if no <filespec> is given to DEBUG. If your are debugging a .COM or .EXE file, however, do not tamper with the program header below address 5CH, or DEBUG will terminate.
2. Do not restart a program after the "Program terminated normally" message is displayed. You must reload the program with the N and L commands for it to run properly.

1.2.2 Method 2: Command Line

To start DEBUG using a command line, type:

```
DEBUG [<filespec> [<arglist>]]
```

For example, if a <filespec> is specified, then the following is a typical command to start DEBUG:

```
DEBUG FILE.EXE
```

DEBUG then loads FILE.EXE into memory starting at 100 hexadecimal in the lowest available segment. The BX:CX registers are loaded with the number of bytes placed into memory.

An <arglist> may be specified if <filespec> is present. The <arglist> is a list of filename parameters and switches that are to be passed to the program <filespec> . Thus, when <filespec> is loaded into memory, it is loaded as if it had been started with the command:

<filespec> <arglist>

Here, <filespec> is the file to be debugged, and the <arglist> is the rest of the command line that is used when <filespec> is invoked and loaded into memory.

CHAPTER 2
COMMANDS

2.1 COMMAND INFORMATION

Each DEBUG command consists of a single letter followed by one or more parameters. Additionally, the control characters and the special editing functions described in the **MS-DOS User's Guide**, apply inside DEBUG.

If a syntax error occurs in a DEBUG command, DEBUG reprints the command line and indicates the error with an up-arrow (^) and the word "error."

For example:

```
dcs:100 cs:110
     ^ error
```

Any combination of uppercase and lowercase letters may be used in commands and parameters.

The DEBUG commands are summarized in Table 2.1 and are described in detail, with examples, following the description of command parameters.

Table 2.1 DEBUG COMMANDS

DEBUG Command	Function
A[<address>]	Assemble
C<range> <address>	Compare
D[<range>]	Dump
E<address> [<list>]	Enter
F<range> <list>	Fill
G[=<address> [<address>. . .]]	Go
H<value> <value>	Hex
I<value>	Input
L[<address> [<drive> <record> <record>]]	Load
M<range> <address>	Move
N<filename> [<filename>]	Name
O<value> <byte>	Output
Q	Quit
R[<register-name>]	Register
S<range> <list>	Search
T[=<address>] [<value>]	Trace
U[<range>]	Unassemble
W[<address> [<drive> <record> <record>]]	Write

2.2 PARAMETERS

All DEBUG commands accept parameters, except the Quit command. Parameters may be separated by delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. Thus, the following commands are equivalent:

```
dcs:100 110
d cs:100 110
d,cs:100,110
```

PARAMETER	DEFINITION
<drive>	A one-digit hexadecimal value to indicate which drive a file will be loaded from or written to. The valid values are 0-3. These values designate the drives as follows: 0=A:, 1=B:, 2=C:, 3=D:.
<byte>	A two-digit hexadecimal value to be placed in or read from an address or register.
<record>	A 1- to 3-digit hexadecimal value used to indicate the logical record number on the disk and the number of disk sectors to be written or loaded. Logical records correspond to sectors. However, their numbering differs since they represent the entire disk space.
<value>	A hexadecimal value up to four digits used to specify a port number or the number of times a command should repeat its functions.
<address>	A two-part designation consisting of either an alphabetic segment register designation or a four-digit segment address plus an offset value. The segment designation or segment address may be omitted, in which case the default segment is used. DS is the default segment for all commands except G, L, T, U, and W, for which the default segment is CS. All numeric values are hexadecimal.

For example:

```
CS:0100
04BA:0100
```

The colon is required between a segment designation (whether numeric or alphabetic) and an offset.

<range<

Two <address>es: e.g., <address> <address>; or one <address>, an L, and a <value>: e.g., <adress> L <value> where <value> is the number of lines the command should operate on, and LB0 is assumed. The last form cannot be used if another hex value follows the <range>, since the hex value would be interpreted as the second <address> of the <range>.
Examples:

```
CS:100 110
CS:100 L 10
CS:100
```

The following is illegal:

```
CS:100 CS:110
       ^ error
```

The limit for <range> is 10 000 hex. To specify a <value> of 10 000 hex within four digits, type 0000 (or 0).

<list>

A series of <byte> values or of <string>s. <list> must be the last parameter on the command line.

Example:

```
fcs:100 42 45 52 54 41
```

<string>

Any number of characters enclosed in quote marks. Quote marks may be either single (’) or double (“). If the delimiter quote marks must appear within a <string>, the quote marks must be doubled. For example, the following strings are legal:

```
’This is a “string” is okay.’
’This is a “string” is okay.’
```

However, this string is illegal:

```
’This is a ’string’ is not.’
```

Similarly, these strings are legal:

```
“This is a ’string’ is okay.”
“This is a ““string”” is okay.”
```

However, this string is illegal:

“This is a “string” is not.”

Note that the double quote marks are not necessary in the following strings:

“This is a ”string” is not necessary.”
’This is a ““string”” is not necessary.’

The ASCII values of the characters in the string are used as a <list> of byte values.

NAME Assemble

PURPOSE Assembles 8086/8087/8088 mnemonics directly into memory.

SYNTAX A[<address>]

COMMENTS If a syntax error is found, DEBUG responds with

```
^Error
```

and redisplays the current assembly address.
All numeric values are hexadecimal and must be entered as 1-4 characters. Prefix mnemonics must be specified in front of the opcode to which they refer. They may also be entered on a separate line.
The segment override mnemonics are CS:, DS:, ES:, and SS:. The mnemonic for the far return is RETF.
String manipulation mnemonics must explicitly state the string size. For example, use MOVSW to move word strings and MOVSB to move byte strings.
The assembler will automatically assemble short, near or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the NEAR or FAR prefix. For example:

```
0100:0500  JMP   502       ; a 2-byte short jump
0100:0502  JMP   NEAR 505  ; a 3-byte near jump
0100:505   JMP   FAR 50A   ; a 5-byte far jump
```

The NEAR prefix may be abbreviated to NE, but the FAR prefix cannot be abbreviated.
DEBUG cannot tell whether some operands refer to a word memory location or to a byte memory location. In this case, the data type must be explicitly stated with the prefix "WORD PTR" or "BYTE PTR". Acceptable abbreviations are "WO" and "BY". For example:

```
NEG    BYTE PTR [128]
DEC    WO [SI]
```

DEBUG also cannot tell whether an operand refers to a memory location or to an immediate operand. DEBUG uses the common convention that operands enclosed in square brackets refer to memory. For example:

```
MOV    AX,21    ; Load AX with 21H
MOV    AX,[21]  ; Load AX with the
                ; contents
                ; of memory location 21H
```

Two popular pseudo-instructions are available with Assemble. The DB opcode will assemble byte values directly into memory. The DW opcode will assemble word values directly into memory. For example:

```
DB     1,2,3,4,“THIS IS AN EXAMPLE”
DB     ’THIS IS A QUOTE: “ ’
DB     “THIS IS A QUOTE: ’ ”

DW     1000,2000,3000,“BACH”
```

Assemble supports all forms of register indirect commands. For example:

```
ADD    BX,34[BP+2].[SI-1]
POP    [BP+DI]
PUSH   [SI]
```

All opcode synonyms are also supported. For example:

```
LOOPZ  100
LOOPE  100

JA     200
JNBE   200
```

For 8087 opcodes, the WAIT or FWAIT must be explicitly specified. For example:

```
FWAIT FADD ST,ST(3) ; This line will assemble
                    ; an FWAIT prefix
LD TBYTE PTR [BX]   ; This line will not
```

NAME Compare

PURPOSE Compares the portion of memory specified by <range> to a portion of the same size beginning at <address>.

SYNTAX C<range> <address>

COMMENTS If the two areas of memory are identical, there is no display and DEBUG returns with the MS-DOS prompt. If there **are** differences, they are displayed in this format:

```
<address1> <byte1> <byte2> <address2>
```

EXAMPLE The following commands have the same effect:

```
C100,1FF   300
     or
C100L100   300
```

Each command compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

NAME Dump

PURPOSE Displays the contents of the specified region of memory.

SYNTAX D[<range>]

COMMENTS If a range of addresses is specified, the contents of the range are displayed. If the D command is typed without parameters, 128 bytes are displayed at the first address (DS:100) after the address displayed by the previous Dump command.

The dump is displayed in two portions: a hexadecimal dump (each byte is shown in hexadecimal value) and an ASCII dump (the bytes are shown in ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line shows 16 bytes with a hyphen between the eighth and ninth bytes. At times, displays are split in this manual to fit them on the page. Each displayed line begins on a 16-byte boundary.

If you type the command:

```
dcs:100 110
```

DEBUG displays the dump in the following format:

```
04BA:0100 42 45 52 54 41 . . . 4E 44 TOM SAWYER
```

If you type the following command:

```
D
```

the display is formatted as described above. Each line of the display begins with an address, incremented by 16 from the address on the previous line. Each subsequent D (typed without parameters) displays the bytes immediately following those last displayed.

If you type the command:

```
DCS:100 L 20
```

the display is formatted as described above, but 20H bytes are displayed.
If then you type the command:

```
DCS:100 115
```

the display is formatted as described above, but all the bytes in the range of lines from 100H to 115H in the CS segment are displayed.

NAME Enter

PURPOSE Enters byte values into memory at the specified <address>.

SYNTAX E<address> [<list>]

COMMENTS If the optional <list> of values is typed, the replacement of byte values occurs automatically. (If an error occurs, no byte values are changed.)

If the <address> is typed without the optional <list>, DEBUG displays the address and its contents, then repeats the address on the next line and wait for your input. At this point, the Enter command waits for you to perform one of the following actions:

1. Replace a byte value with a value you type. Simply type the value after the current value. If the value typed in is not a legal hexadecimal value or if more than two digits are typed, the illegal or extra character is not echoed.
2. Press the <SPACE> bar to advance to the next byte. To change the value, simply type the new value as described in (1.) above. If you space beyond an 8-byte boundary, DEBUG starts a new display line with the address displayed at the beginning.
3. Type a hyphen (-) to return to the preceding byte. If you decide to change a byte behind the current position, typing the hyphen returns the current position to the previous byte. When the hyphen is typed, a new line is started with the address and its byte value displayed.
4. Press the <NEW LINE> key to terminate the Enter command. The <NEW LINE> key may be pressed at any byte position.

EXAMPLE

Assume that the following command is typed:

```
ECS:100
```

DEBUG displays:

```
04BA:0100   EB.-
```

To change this value to 41, type 41 as shown:

```
04BA:0100   EB.41-
```

To step through the subsequent bytes, press the <SPACE> bar to see:

```
04BA:0100    EB.41    10.    00.    BC.-
```

To change BC to 42:

```
04BA:0100    EB.41    10.    00.    BC.42-
```

Now, realizing that 10 should be 6F, type the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F:

```
04BA:0100     EB.41     10.     00.     BC.42-
04BA:0102     00.--
04BA:0101     10.6F-
```

Pressing the <NEW LINE> key ends the Enter command and retunrs to the DEBUG command level.

NAME Fill

PURPOSE Fills the addresses in the <range> with the values in the <list>.

SYNTAX F<range> <list>

COMMENTS If the <range> contains more bytes than the number of values in the <list>, the <list> will be used repeatedly until all bytes in the <range> are filled. If the <list> contains more values than the number of bytes in the <range>, the extra values in the <list> will be ignored. If any of the memory in the <range> is not valid (bad or nonexistent), the error will occur in all succeeding locations.

EXAMPLE Assume that the following command is typed:

```
F04BA:100 L 100 42 45 52 54 41
```

DEBUG fills memory locations 04BA:100 through 04BA:1FF with the bytes specified. The five values are repeated until all 100H bytes are filled.

NAME Go

PURPOSE Executes the program currently in memory.

SYNTAX

```
G [=<address> [<address>. . .]]
```

COMMENTS If only the Go command is typed, the program executes as if the program had run outside DEBUG.

If = <address> is set, execution begins at the address specified. The equal sign (=) is required, so that DEBUG can distinguish the start = <address> from the breakpoint <address>es.

With the other optional addresses set, execution stops at the first <address> encountered, regardless of that address' position in the list of addresses to halt execution or program branching. When program execution reaches a breakpoint, the registers, flags, and decoded instruction are displayed for the last instruction executed. (The result is the same as if you had typed the Register command for the breakpoint address.)

Up to ten breakpoints may be set. Breakpoints may be set only at addresses containing the first byte of an 8086 opcode. If more than ten breakpoints are set, DEBUG returns the BP Error message.

The user stack pointer must be valid and have 6 bytes available for this command. The G command uses an IRET instruction to cause a jump to the program under test. The user stack pointer is set, and the user flags, Code Segment register, and Instruction Pointer are pushed on the user stack. (Thus, if the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es).

When an instruction with the breakpoint code is encountered, all breakpoint addresses are restored to their original instructions. If execution is not halted at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

EXAMPLE Assume that the following command is typed:

```
GCS:7550
```

The program currently in memory executes up to the address 7550 in the CS segment. DEBUG then displays registers and flags, after which the Go command is terminated.

After a breakpoint has been encountered, if you type the Go command again, then the program executes just as if you had typed the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint rather than at the usual start address.

NAME Hex

PURPOSE Performs hexadecimal arithmetic on the two parameters specified.

SYNTAX H<value> <value>

COMMENTS First, DEBUG adds the two parameters, then subtracts the second parameter from the first. The results of the arithmetic are displayed on one line; first the sum, then the difference.

EXAMPLE Assume that the following command is typed:

```
H19F 10A
```

DEBUG performs the calculations and then displays the result:

```
02A9   0095
```

NAME Input

PURPOSE Inputs and displays one byte from the port specified by <value>.

SYNTAX I<value>

COMMENTS A 16-bit port address is allowed.

EXAMPLE Assume that you type the following command:

```
I2F8
```

Assume also that the byte at the port is 42H. DEBUG inputs the byte and displays the value:

```
42
```

NAME Load

PURPOSE Loads a file into memory.

SYNTAX L[<address> [<drive> <record> <record>]]

COMMENTS Set BX:CX to the number of bytes read. The file must have been named either when DEBUG was started or with the N command. Both the DEBUG invocation and the N command format a filename properly in the normal format of a file control block at CS:5C.

If the L command is typed without any parameters, DEBUG loads the file into memory beginning at address CS:100 and sets BX:CX to the number of bytes loaded. If the L command is typed with an address parameter, loading begins at the memory <address> specified. If L is typed with all parameters, absolute disk sectors are loaded, not a file. The <record>s are taken from the <drive> specified (the drive designation is numeric here–0=A:, 1=8:, 2=C:, etc.); DEBUG begins loading with the first <record> specified, and continues until the number of sectors specified in the second <record> have been loaded.

EXAMPLE Assume that the following commands are typed:

```
A>DEBUG
-NFILE.COM
```

Now, to load FILE.COM, type:

```
L
```

DEBUG loads the file and then displays the DEBUG prompt. Assume that you want to load only portions of a file or certain records from a disk. To do this, type:

```
L04BA:100 2 0F 6D
```

DEBUG then loads 109 (6D hex) records beginning with logical record number 15 into memory beginning at address 04BA:0100. When the records have been loaded, DEBUG simply returns the - prompt.

If the file has a .EXE extension, it is relocated to the load address specified in the header of the .EXE file: the <address> parameter is always ignored for .EXE files. The header itself is stripped off the .EXE file before it is loaded into memory. Thus the size of an .EXE file on disk will differ from its size in memory.

If the file named by the Name command or specified when DEBUG is started is a .HEX file, then typing the L command with no parameters causes DEBUG to load the file beginning at the address specified in the .HEX file. If the L command includes the option <address>, DEBUG adds the <address> specified in the L command to the address found in the .HEX file to determine the start address for loading the file.

NAME Move

PURPOSE Moves the block of memory specified by <range> to the location beginning at the <address> specified.

SYNTAX M<range> <address>

COMMENTS Overlapping moves (i.e., moves where part of the block overlaps some of the current addresses) are always performed without loss of data. Addresses that could be overwritten are moved first. The sequence for moves from higher addresses to lower addresses is to move the data beginning at the block's lowest address and then to work towards the highest. The sequence for moves from lower addresses to higher addresses is to move the data beginning at the block's highest address and to work towards the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data there before the move will remain. The M command copies the data from one area into another, in the sequence described, and writes over the new addresses. This is why the sequence of the move is important.

EXAMPLE Assume that you type:

```
MCS:100 110 CS:500
```

DEBUG first moves address CS:110 to address CS:510, then CS:10F to CS:50F, and so on until CS:100 is moved to CS:500. You should type the D command, using the <address> typed for the M command, to review the results of the move.

NAME Name

PURPOSE Sets filenames.

SYNTAX N<filename> [<filename> . . .]

COMMENTS The Name command performs two functions. First, Name is used to assign a filename for a later Load or Write command. Thus, if you start DEBUG without naming any file to be debugged, then the N<filename> command must be typed before a file can be loaded. Second, Name is used to assign filename parameters to the file being debugged. In this case, Name accepts a list of parameters that are used by the file being debugged.

These two functions overlap. Consider the following set of DEBUG commands:

```
-NFILE1.EXE
-L
-G
```

Because of the effects of the Name command, Name will perform the following steps:

1. (N)ame assigns the filename FILE1.EXE to the filename to be used in any later Load or Write commands.
2. (N)ame also assigns the filename FILE1.EXE to the first filename parameter used by any program that is later debugged.
3. (L)oad loads FILE1.EXE into memory.
4. (G)o causes FILE1.EXE to be executed with FILE1.EXE as the single filename parameter (that is, FILE1.EXE is executed as if FILE1.EXE had been typed at the command level).

A more useful chain of commands might look like this:

```
-NFILE1.EXE
-L
-NFILE2.DAT FILE3.DAT
-G
```

Here, Name sets FILE1.EXE as the filename for the subsequent Load command. The Load command loads FILE1.EXE into memory, and then the Name command is used again, this time to specify the parameters to be used by FILE1.EXE. Finally, when the Go command is executed, FILE1.EXE is executed as if FILE1 FILE2.DAT FILE3.DAT had been typed at the MS-DOS command level. Note that if a Write command were executed at this point, then FILE1.EXE - the file being debugged - would be saved with the name FILE2.DAT! To avoid such undesired results, you should always execute a Name command before either a Load or a Write.

There are four regions of memory that can be affected by the Name command:

CS:5C	FCB for file 1
CS:6C	FCB for file 2
CS:80	Count of characters
CS:81	All characters typed

A File Control Block (FCB) for the first filename parameter given to the Name command is set up at CS:5C. If a second filename parameter is typed, then an FCB is set up for it beginning at CS:6C. The number of characters typed in the Name command exclusive of the first character, "N") is given at location CS:80. The actual stream of characters given by the Name command (again, exclusive of the letter "N") begins at CS:81. Note that this stream of characters may contain switches and delimiters that would be legal in any command typed at the MS-DOS command level.

EXAMPLE

A typical use of the Name command is:

```
DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this case, the Go command executes the file in memory as if the following command line had been typed:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging therefore reflect a normal runtime environment for PROG.COM.

NAME Output

PURPOSE Sends the <byte> specified to the output port specified by <value>.

SYNTAX 0<value> <byte>

COMMENTS A 16-bit port address is allowed.

EXAMPLE Type:

```
02F8 4F
```

DEBUG outputs the byte value 4F to output port 2F8.

NAME Quit

PURPOSE Terminates the DEBUG utility.

SYNTAX Q

COMMENTS The Q command takes no parameters and exits DEBUG without saving the file currently being operated on. You are returned to the MS-DOS command level.

EXAMPLE To end the debugging session, type:

```
Q <NEW LINE>
```

DEBUG has been terminated, and control returns to the MS-DOS command level.

NAME Register

PURPOSE Displays the contents of one or more CPU registers.

SYNTAX R[<register-name>]

COMMENTS If no <register-name> is typed, the R command dumps the register save area and displays the contents of all registers and flags.

If a register name is typed, the 16-byte value of that register is displayed in hexadecimal, and then a colon appears as a prompt. You then either type a <value> to change the register, or simply press the <NEW LINE> key if no change is wanted.

The only valid <register-name>s are:

AX	BP	SS	
BX	SI	CS	
CX	DI	IP	(IP and PC both refer to
DX	DS	PC	the Instruction Pointer.)
SP	ES	F	

Any other entry for <register-name> results in a BR Error message.

If F is entered as the <register-name>, DEBUG displays each flag with a two-character alphabetic code. To alter any flag, type the opposite two-letter code. The flags are either set or cleared.

The flags are listed below with their codes for SET and CLEAR:

FLAG NAME	SET	CLEAR
Overflow	OV	NV
Direction	DN Decrement	UP Increment
Interrupt	EI Enabled	DI Disabled
Sign	NG Negative	PL Plus
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE Even	PO Odd
Carry	CY	NC

Whenever you type the command RF, the flags are displayed in the order shown above in a row at the beginning of a line. At the end of the list of flags, DEBUG displays a hyphen (-). You may enter new flag values as alphabetic pairs. The new flag values can be entered in any order. You do not have to leave spaces between the flag entries. To exit the R command, press the <NEW LINE> key. Flags for which new values were not entered remain unchanged.

If more than one value is entered for a flag, DEBUG returns a DF Error message. If you enter a flag code other than those shown above, DEBUG returns a BF Error message. In both cases, the flags up to the error in the list are changed; flags at and after the error are not.

At startup, the segment registers are set to the bottom of free memory, the Instruction Pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

EXAMPLE

Type:

```
R
```

DEBUG displays all registers, flags, and the decoded instruction for the current location. If the location is CS:11A, then the display will look similar to this:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D
BP=0000 SI=005C DI=0000 DS=04BA ES=04BA
SS=04BA CS=04BA IP=011A
NV UP DI NG NZ AC PE NC
04BA:011A   CD21        INT     21
```

If you type:

```
RF
```

DEBUG will display the flags:

```
NV UP DI NG NZ AC PE NC - -
```

Now, type any valid flag designation, in any order, with or without spaces.

For example:

```
NV UP DI NG NZ AC PE NC - PLEICY<NEW LINE>
```

DEBUG responds only with the DEBUG prompt. To see the changes, type either the R or RF command:

```
RF
NV UP EI PL NZ AC PE CY - -
```

Press <NEW LINE> to leave the flags this way, or to specify different flag values.

NAME Search

PURPOSE Searches the <range< specified for the <list> of bytes specified.

SYNTAX S<range> <list>

COMMENTS The <list> may contain one or more bytes, each separated by a space or comma. If the <list> contains more than one byte, only the first address of the byte string is returned. If the <list> contains only one byte, all addresses of the byte in the <range> are displayed.

EXAMPLE If you type:

```
SCS:100 110 41
```

DEBUG will display a response similar to this:

```
04BA:0104
04BA:010D
-type:
```

NAME Trace

PURPOSE Executes one instruction and displays the contents of all registers and flags, and the decoded instruction.

SYNTAX T[=<address>] [<value>]

COMMENTS If the optional =<address> is typed, tracing occurs at the =<address> specified. The optional <value> causes DEBUG to execute and trace the number of steps specified by <value>.

The T command uses the hardware trace mode of the 8086 or 8088 microprocessor. Consequently, you may also trace instructions stored in ROM (Read Only Memory).

EXAMPLE TYPE:

```
T
```

DEBUG returns a display of the registers, flags, and decoded instruction for that one instruction. Assume that the current position is 04BA:011A; DEBUG might return the display:

```
AX=0E00 BX=00FF CS=0007 DX=01FF SP=039D
BP=0000 SI=005C DI=0000 DS=04BA ES=04BA
SS=04BA CS=04BA IP=011A
NV UP DI NG NZ AC PE NC
04BA:011A   CD21        INT     21
```

If you type

```
T=011A 10
```

DEBUG executes sixteen (10 hex) instructions beginning at 011A in the current segment, and then displays all registers and flags for each instruction as it is executed. The display scrolls away until the last instruction is executed. Then the display stops, and you can see the register and flag values for the last few instructions performed. Remember that <CONTROL-S> suspends the display at any point, so that you can study the registers and flags for any instruction.

NAME Unassemble

PURPOSE Disassembles bytes and displays the source statements that correspond to them, with addresses and byte values.

SYNTAX U[<range>]

COMMENTS The display of disassembled code looks like a listing for an assembled file. If you type the U command without parameters, 20 hexadecimal bytes are disassembled at the first address after that displayed by the previous Unassemble command. If you type the U command with the <range> parameter, then DEBUG disassembles all bytes in the range. If the <range> is given as an <address> only, then 20H bytes are disassembled instead of 80H.

EXAMPLE Type:

```
U04BA:100 L10
```

DEBUG disassembles 16 bytes beginning at address 04BA:0100:

```
04BA:0100  206472  AND   [SI+72],AH
04BA:0103  69      DB    69
04BA:0104  7665    JBE   016B
04BA:0106  207370  AND   [BP+DI+70],DH
04BA:0109  65      DB    65
04BA:010A  63      DB    63
04BA:010B  69      DB    69
04BA:010C  66      DB    66
04BA:010D  69      DB    69
04BA:010E  63      DB    63
04BA:010F  61      DB    61
```

If you type

```
004ba:0100 0108
```

The display will show:

```
04BA:0100 206472  AND  [SI+72],AH
04BA:0103 69      DB   69
04BA:0104 7665    JBE  016B
04BA:0106 207370  AND  [BP+DI+70],DH
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. The U command can be typed for the changed locations, the new instructions viewed, and the disassembled code used to edit the source file.

NAME	Write
PURPOSE	Wirtes the file being debugged to a disk file.
SYNTAX	W[<address> [<drive> <record> <records>]]
COMMENTS	If you type W with no parameters, BX:CX must already be set to the number of bytes to be written; the file is written beginning from CS:100. If the W command is typed with just an address, then the file is written beginning at that address. If a G or T command has been used, BX:CX must be reset before using the Write command without parameters. Note that if a file is loaded and modified, the name, length, and starting address are all set correctly to save the modified file (as long as the length has not changed). The file must have been named either with the DEBUG invocation command or with the N command (refer to the Name command earlier in this manual). Both the DEBUG invocation and the N command format a filename properly in the normal format of a file control block at CS:5C. If the W command is typed with parameters, the write begins from the memory address specified; the file is written to the <drive> specified (the drive designation is numeric here-0=A:, 1=B:, 2=C:, etc.); DEBUG writes the file beginning at the logical record number specified by the first <record>; DEBUG continues to write the file until the number of sectors specified in the second <record> have been written.

WARNING

Writing to absolute sectors is **EXTREMELY** dangerous because the process bypasses the file handler.

EXAMPLE Type:

```
W
```

DEBUG will write the file to disk and then display the DEBUG prompt. Two examples are shown below.

```
W
--
```

```
WCS:100 1 37 2B
```

DEBUG writes out the contents of memory, beginning with the address CS:100 to the disk in drive B:. The data written out starts in disk logical record number 37H and consists of 2BH records. When the write is complete, DEBUG displays the prompt:

```
WCS:100 1 37 2B
--
```

2.3 ERROR MESSAGES

During the DEBUG session, you may receive any of the following error messages. Each error terminates the DEBUG command under which it occurred, but does not terminate DEBUG itself.

ERROR CODE	DEFINITION
BF	Bad flag You attempted to alter a flag, but the characters typed were not one of the acceptable pairs of flag values. See the Register command for the list of acceptable flag entries.
BP	Too many breakpoints You specified more than ten breakpoints as parameters to the G command. Retype the Go command with ten or fewer breakpoints.
BR	Bad register You typed the R command with an invalid register name. See the Register command for the list of valid register names.
DF	Double flag You typed two values for one flag. You may specify a flag value only once per RF command.