IBM

Programming Series

Program No.5601-SBS

110123-0

JX PC DOS Version 2.10 Techical Reference

IBM·JX
personal computer

First Edition (August 1985)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

It is possible that this material may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or Services in your country.

Preface

Read This First

This technical reference manual covers topics for the more experienced DOS users, system programmers, and those who will be developing their own applications.

Organization of This Manual

This manual has 9 chapters.

Chapter 1 contains general technical information.

Chapter 2 contains detailed information about using extended screen and keyboard functions.

Chapter 3 contains detailed information about device drivers.

Chapter 4 describes allocation of space on a disk.

Chapter 5 describes the system interrupts and function calls.

Chapter 6 describes control blocks and work areas, including a Memory Map, Program Segment, and File Control Block.

Chapter 7 explains how to execute commands from within an application.

Chapter 8 contains technical information about DOS support of fixed disks.

Chapter 9 contains detailed information about .EXE file structure.

Chapter 10 contains information about DOS memory management.

Contents

Chapter 1. DOS Technical Information 1-1
DOS Structure 1-3
DOS Initialization 1-4
The Command Processor 1-5
Available DOS Functions 1-6
File Management Notes 1-6
The Disk Transfer Area (DTA) 1-7
Error Trapping 1-8

Chapter 2. Using Extended Screen and Keyboard Control 2-1
Introduction 2-3
Cursor Control 2-4
Cursor Position 2-4
Cursor Up 2-4
Cursor Down 2-5
Cursor Forward 2-5
Cursor Backward 2-5
Horizontal and Vertical Position 2-6
Device Status Report 2-6
Cursor Position Report 2-6
Save Cursor Position 2-7
Restore Cursor Position 2-7
Erasing 2-8
Erase in Display 2-8
Erase in Line 2-8
Mode of Operation 2-9
Set Graphics Rendition 2-9
Set Mode 2-10
Reset Mode 2-10
Keyboard Key Reassignment 2-11

Chapter 3. Installable Device Drivers 3-1
Introduction 3-3
Device Driver Format 3-4
Types of Devices 3-4
Device Header 3-6
Creating a Device Driver 3-9
Installation of Device Drivers 3-10
Request Header 3-11
Unit Code 3-11
Command Code 3-12
Status Word 3-13
Function Call Parameters 3-16
MEDIA Descriptor Byte 3-21
The Time of Day Device Driver 3-26
The Memory Device Driver 3-26

Chapter 4. DOS Disk Allocation 4-1
DOS Disk Directory 4-4
DOS File Allocation Table 4-8
How to Use the File Allocation Table 4-10

Chapter 5. DOS Interrupts and Function Calls 5-1
Interrupts 5-3
Function Calls 5-13
Invoking DOS Functions 5-16

Chapter 6. DOS Control Blocks and Work Areas 6-1
DOS Memory Map 6-3
DOS Program Segment 6-5
Program Segment Prefix 6-9
File Control Block 6-11

Chapter 7. Executing Commands from Within an Application 7-1

Chapter 8. Fixed Disk Information 8-1
Fixed Disk Architecture 8-3
System Initialization 8-4
Boot Record/Partition Table 8-6
Technical Information 8-8

Chapter 9. EXE File Structure
and Loading 9-1

Chapter 10. DOS Memory Management 10-1

Index X-1

Chapter 1. DOS Technical Reference Information

Contents

DOS Structure 1-3
DOS Initialization 1-4
The Command Processor 1-5
Available DOS Functions 1-6
File Management Notes 1-6
The Disk Transfer Area (DTA) 1-7
Error Trapping 1-8

This book is intended to supply technically oriented users with information about the structure, facilities, and program interfaces of DOS. It is assumed that the reader is familiar with the 8088 architecture, interrupt mechanism, and instruction set.

DOS Structure

DOS consists of the following four components:

1. The boot record resides on track 0, sector 1, side 0 of every disk formatted by the FORMAT command. It is put on all disks in order to produce an error message if you try to start up the system with a non-DOS diskette in drive A. For fixed disks, it resides on the first sector (sector 1, head 0) of the first cylinder of the DOS partition.

2. The Read-Only Memory (ROM) BIOS interface module (file IBMBIO.COM) provides a low-level interface to the ROM BIOS device routines.

3. The DOS program itself (file IBMDOS.COM) provides a high-level interface for user programs. It consists of file management routines, data blocking/deblocking for the disk routines, and a variety of built-in functions easily accessible by user programs. (Refer to Chapter 5.)

   When these function routines are invoked by a user program, they accept high-level information via register and control block contents, then (for device operations) translate the requirement into one or more calls to IBMBIO to complete the request.

4. The command processor, COMMAND.COM.

DOS Initialization

When the system is started (either System Reset or power ON with the DOS diskette in drive A), the boot record is read into memory and given control. It checks the directory to assure that the first two files listed are IBMBIO.COM and IBMDOS.COM, in that order. (An error message is issued if not.) These two files are then read into memory. (IBMBIO.COM must be the first file in the directory, and its sectors must be contiguous.)

The initialization code in IBMBIO.COM determines equipment status, resets the disk system, initializes the attached devices, causes device drivers to be loaded, and sets the low-numbered interrupt vectors. It then relocates IBMDOS.COM downward and calls the first byte of DOS.

As in IBMBIO.COM, offset 0 in DOS contains a jump to its initialization code, which will later be overlaid by a data area and the command processor. DOS initializes its internal working tables, initializes interrupt vectors for interrupts hex 20 through hex 27 and builds a Program Segment Prefix (see Chapter 6) for COMMAND.COM at the lowest available segment, then returns to IBMBIO.COM.

The last remaining task of initialization is for IBMBIO.COM to load COMMAND.COM at the location set up by DOS initialization. IBMBIO.COM then passes control to the first byte of COMMAND.

The Command Processor

The command processor supplied with DOS (file COMMAND.COM) consists of four distinctly separate parts:

- A resident portion resides in memory immediately following IBMDOS.COM and its data area. This portion contains routines to process interrupt types hex 22 (terminate address), hex 23 (CTRL-BREAK handler), and hex 24 (critical error handling), as well as a routine to reload the transient portion if needed. (When a program terminates, a checksum methodology determines if the program had caused the transient portion to be overlaid. If so, it is reloaded.) Note that all standard DOS error handling is done within this portion of COMMAND. This includes displaying error messages and interpreting the reply of Abort, Retry, or Ignore. (See message Disk error reading drive *x* in Appendix A of the *Disk Operating System Reference* manual.)

- An initialization portion follows the resident portion and is given control during startup. This section contains the AUTOEXEC file processor setup routine. The initialization portion determines the segment address at which programs can be loaded. It is overlaid by the first program COMMAND loads because it's no longer needed.

- A transient portion is loaded at the high end of memory. This is (portion 3) the command processor itself, containing all of the internal command processors, the batch file processor, and (portion 4) a routine to load and execute external commands (files with filename extensions of .COM or .EXE). This "loader" is at the highest end of memory, and is invoked by the EXEC function call to load programs.

Portion 3 of COMMAND produces the system prompt (such as A>), reads the command from the keyboard (or batch file) and causes it to be executed. For external commands, it builds a command line and issues an EXEC function call to load and transfer control to the program.

Chapter 6 contains detailed information describing the conditions in effect when a program is given control by EXEC.

Available DOS Functions

DOS provides a significant number of functions to user programs, all available through issuance of a set of interrupt codes. There are routines for keyboard input (with and without echo and Ctrl-Break detection), console and printer output, constructing file control blocks, memory management, date and time functions, and a variety of disk, directory, and file handling functions. See Chapter 5 "DOS Interrupts and Function Calls" for detailed information.

File Management Notes

Through the INT 21 (function call) mechanism, DOS provides methods to create, read, write, rename, and erase files. Files are not necessarily written sequentially on disk–space is allocated as it is needed, and the first location available on the disk is allocated as the next location for a file being written. Therefore, if considerable file creation and erasure activity has taken place, newly created files will probably *not* be written in sequential sectors.

However, due to the mapping (chaining) of file space via the File Allocation Table, and the function calls available, any file can be used in either a sequential or random manner.

There are two sets of function calls that support file management. The new, extended set of calls are the preferred method (functions 39 through 57). Through these calls, sequential and random file accesses are simpler than using the traditional (FCB oriented) set of calls. The FCB calls continue to function as in the past: By using the current block and current record fields of the FCB, and the sequential disk read or write functions, you can make the file appear sequential– DOS will do the calculations necessary to locate the proper sectors on the disk. On the other hand, by using the random record field, and random disk functions, you can cause any record in the file to be accessed *directly*– again, DOS will locate the correct sectors on the disk for you.

Space is allocated in increments called *clusters*. For single sided diskettes, this unit of allocation is one sector; for dual sided diskettes, each cluster is two consecutive sectors in length. The cluster size of a fixed disk is determined at FORMAT time, and is based on the size of the DOS partition.

The Disk Transfer Area (DTA)

The Disk Transfer Area (also commonly called *buffer*) is the memory area DOS will use to contain the data for all file reads and writes that are performed with the traditional (FCB) set of function calls. This area can be at any location within memory, and should be set by your program. (See function call hex 1A.)

Only one DTA can be in effect at a time, so it is the program's responsibility to inform DOS what memory location to use *before* using any disk read or write functions. Once set, DOS continues to use that area for all disk operations until another function call hex 1A is issued to define a new DTA. When a program is given control by COMMAND, a default DTA has already been established at hex 80 into the program's Program Segment Prefix, large enough to hold 128 bytes.

When using the extended file management function calls, you specify a buffer address when you issue the read or write call. There is no need to set a DTA address.

Error Trapping

DOS provides a method by which a program can receive control whenever a disk or device read/write error occurs, or when a bad memory image of the file allocation table is detected. When these events occur, DOS executes an INT hex 24 to pass control to the error handler. The default error handler resides in COMMAND.COM, but any program can establish its own by setting the INT hex 24 vector to point to the new error handler. DOS provides error information via the registers and provides Abort, Retry, or Ignore support via return codes. (Refer to Chapter 5, "DOS Interrupts and Function Calls".)

Chapter 2. Using Extended Screen and Keyboard Control

Contents

Introduction 2-3

Cursor Control 2-4
- Cursor Position 2-4
- Cursor Up 2-4
- Cursor Down 2-5
- Cursor Forward 2-5
- Cursor Backward 2-5
- Horizontal and Vertical Position 2-6
- Device Status Report 2-6
- Cursor Position Report 2-6
- Save Cursor Position 2-7
- Restore Cursor Position 2-7

Erasing 2-8
- Erase in Display 2-8
- Erase in Line 2-8

Mode of Operation 2-9
- Set Graphics Rendition 2-9
- Set Mode 2-10
- Reset Mode 2-10

Keyboard Key Reassignment 2-11

Introduction

With DOS Version 2.10 you can issue special character sequences from within your program that can be used to control screen cursor positioning. You can also reassign the meaning of any key in the keyboard.

Notes:

1. The control sequences defined below are valid when issued through any DOS function calls, that can write to the standard output device. These control sequences require the presence of the extended screen and keyboard control device driver. This can be accomplished by placing the command:

   ```
   DEVICE=ANSI.SYS
   ```

   in your CONFIG.SYS (configuration) file. Note that the size of DOS in memory will be increased by the size of the ANSI.SYS program.

2. The default value is used when no explicit value, or a value of zero, is specified.

3. # – Numeric Parameter. A decimal number specified with ASCII characters.

4. In the control sequences described below, ESC is the 1 byte code for ESC (hex 1B), *not* the three characters "ESC." For example, ESC [2;10H could be created under DEBUG as follows:

   ```
   e200 1B "[2;10H"
   ```

Cursor Control

Cursor Position

CUP	Function
ESC [#;#H	Moves the cursor to the position specified by the parameters. The first parameter specifies the line number and the second parameter specifies the column number. The default value is one. If no parameter is given, the cursor is moved to the home position.

Cursor Up

CUU	Function
ESC [#A	Moves the cursor up one line without changing columns. The value of # determines the number of lines moved. The default value for # is one. This sequence is ignored if the cursor is already on the top line.

Cursor Down

CUD	Function
ESC [#B	Moves the cursor down one line without changing columns. The value of # determines the number of lines moved. The default value for # is one. The sequence is ignored if the cursor is already on the bottom line.

Cursor Forward

CUF	Function
ESC [#C	Moves the cursor forward one column without changing lines. The value of # determines the number of columns moved. The default value for # is one. This sequence is ignored if the cursor is already in the rightmost column.

Cursor Backward

CUB	Function
ESC [#D	Moves the cursor back one column without changing lines. The value of # determines the number of columns moved. The default value for # is one. This sequence is ignored if the cursor is already in the leftmost column.

Horizontal and Vertical Position

HVP	Function
ESC [#;#f	Moves the cursor to the position specified by the parameters. The first parameter specifies the line number and the second parameter specifies the column number. The default value is one. If no parameter is given, the cursor is moved to the home position (same as CUP).

Device Status Report

DSR	Function
ESC [6n	The console driver will output a CPR sequence on receipt of DSR (see below).

Cursor Position Report

CPR	Function
ESC [#;#R	The CPR sequence reports the current cursor position through the standard input device. The first parameter specifies the current line and the second parameter specifies the current column.

Save Cursor Position

SCP	Function
ESC [s	The current cursor position is saved. This cursor position can be restored with the RCP sequence.

Restore Cursor Position

RCP	Function
ESC [u	Restores the cursor to the value it had when the console driver received the SCP sequence.

Erasing

Erase in Display

ED	Function
ESC [2J	Erases all of the screen and the cursor goes to the home position.

Erase in Line

EL	Function
ESC [k	Erases from the cursor to the end of the line and includes the cursor position.

Set Graphics Rendition

SGR	Function
ESC [#;...;#m	Sets the character attribute specified by the parameter(s). All following characters will have the attribute according to the parameter(s) until the next occurrence of SGR.

Parameter	Meaning
0	All attributes Off (normal white on black)
1	Bold On (high intensity)
4	Underscore On (IBM Monochrome Display only)
5	Blink On
7	Reverse video On
8	Cancelled On (invisible)
30	Black foreground
31	Red foreground
32	Green foreground
33	Yellow foreground
34	Blue foreground
35	Magenta foreground
36	Cyan foreground
37	White foreground
40	Black background
41	Red background
42	Green background
43	Yellow background
44	Blue background
45	Magenta background
46	Cyan background
47	White background

Set Mode

SM	Function
ESC [=#h or ESC [=h or ESC [=0h or ESC [?7h	Invokes the screen width or type specified by the parameter. Parameter Meaning 0 40x25 black and white 1 40x25 color 2 80x25 black and white 3 80x25 color 4 320x200 color 5 320x200 black and white 6 640x200 black and white 7 wrap at end of line (typing past end-of-line results in new line)

Reset Mode

RM	Function
ESC [=#l or ESC [=l or ESC [=0l or ESC [?7l	Parameters are the same as SM (Set Mode) except that parameter 7 will reset wrap at end-of-line mode (characters past end-of-line are thrown away).

Keyboard Key Reassignment

The control sequence is	Function
ESC [#;#;...#p or **ESC ["string";p** or **ESC [#;"string";#;#;"string";#p** or any other combination of strings and decimal numbers	The first ASCII code in the control sequence defines which code is being mapped. The remaining numbers define the sequence of ASCII codes generated when this key is intercepted. However, if the first code in the sequence is zero (NUL) then the first and second code make up an extended ASCII re-definition. The IBM Personal Computer JX *BASIC* manual has a list of all the ASCII and extended ASCII codes).

Here are some examples:

1. Reassign the Q and q key to the A and a key (and the other way as well):

   ESC [65;81p A becomes Q
   ESC [97;113p a becomes q
   ESC [81P;65p Q becomes A
   ESC [113;97p q becomes a

2. Reassign the F10 key to a DIR command followed by a carriage return:

   ESC [0;68;"dir";13p

   The 0;68 is the extended ASCII code for the F10 key. 13 decimal is a carriage return.

3. The following assembly language program reassigns the PF10 key to a DIR B: command followed by a carriage return.

```
        TITLE SETANSI.ASM - SET F10 TO STRING FOR ANSI.SYS
CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME CS:CSEG,DS:CSEG
;
        ORG     50H
DOS     LABEL   FAR                     ;INT 21H        RET FAR
        ORG     100H
ENTPT:  JMP     SHORT START
STRING  DB      27,'[0;68;"SDIR B:";13p' ;REDEFINE F10 KEY



STRSIZ  EQU     $-STRING                ;LENGTH OF ABOVE MESSAGE
HANDLE  EQU     1                       ;PRE-DEFINED FILE HANDLE FOR STANDARD OUTPUT
;
;                       VECTOR TO DOS CALL ROUTINE
DOSVECTOR LABEL DWORD
DOSOFF  DW      DOS                     ; OFFSET
DOSSEG  DW      0                       ; SEGMENT (TO BE FILLED)
;
START   PROC    NEAR
        MOV     AX,CS                   ;FIND WHERE THIS CODE IS
        MOV     DOSSEG,AX               ;SET VECTOR ACCORDINGLY
;(this is done so this code becomes self-relocatable, suitable for .com file)
;
        MOV     BX,HANDLE               ;STANDARD OUTPUT DEVICE
        MOV     CX,STRSIZ               ;GET SIZE OF TEXT TO BE SENT
        MOV     DX,OFFSET STRING        ;PASS OFFSET OF STRING TO BE SENT
        MOV     AH,40H                  ;FUNCTION="WRITE TO DEVICE"
        CALL    DOSVECTOR               ;CALL DOS
        RET                             ;RETURN TO DOS
START   ENDP
CSEG    ENDS
        END     ENTPT
```

Chapter 3. Installable Device Drivers

Contents

Introduction 3-3

Device Driver Format 3-4
- Types of Devices 3-4
  - Character Devices 3-5
  - Block Devices 3-5
- Device Header 3-6
  - Next Device Header Field 3-6
  - Attribute Field 3-7
  - Strategy and Interrupt Routines 3-8
  - Name Field 3-8

Creating a Device Driver 3-9

Installation of Device Drivers 3-10

Request Header 3-11
- Unit Code 3-11
- Command Code 3-12
- Status Word 3-13
- Function Call Parameters 3-16
  - INIT 3-17
  - MEDIA CHECK 3-18
  - BUILD BPB (BIOS Parameter Block) 3-18
- MEDIA Descriptor Byte 3-21
  - INPUT or OUTPUT 3-22
  - Non Destructive Input No Wait 3-24
  - STATUS 3-24
  - FLUSH 3-25

The Time of Day Device Driver 3-26

The Memory Device Driver 3-26

Introduction

The DOS Version 2.10 device interface links the devices together in a chain. This allows new device drivers for optional devices to be added to DOS.

Device Driver Format

A device driver is a .COM file with all of the code in it to implement the device. In addition it has a special header at the front of it that identifies it as a device, defines the strategy and interrupt entry points, and defines various attributes of the device.

> Note: For device drivers, the .COM file must not use the ORG 100H. Because it does not use the program segment prefix, the device driver is simply loaded; therefore, the .COM file must have an orgin of zero (ORG 0 or no ORG statement).

Types of Devices

There are two basic types of devices:

- Character devices
- Block devices

Character Devices

These are devices that are designed to do character I/O in a serial manner like CON, AUX, and PRN. These devices have names like CON, AUX, CLOCK$, and you can open channels (handles or FCBs) to do input and output to them.

> Note: Because character devices have only one name, they can support only one device.

Block Devices

These devices are the "fixed disk or diskette drives" on the system, they can do random I/O in pieces called blocks (usually the physical sector size of the disk). These devices are not *named* as the character devices are, and cannot be *opened* directly. Instead they are *mapped* via the drive letters (A, B, C, etc.). Block devices can have units within them. In this way, a single block driver can be responsible for one or more disk or diskette drives. For example, block device driver ALPHA can be responsible for drives A, B, C and D. This means that ALPHA has four units defined and therefore takes up four drive letters. The way the drive units and drive letters correspond is determined by the position of the driver in the chain of all drivers. For example, if device driver ALPHA is the first block driver in the device chain, and it has defined four units, then those units will be A, B, C and D. If BETA is the second block driver, and it defines three units, then those units will be E, F and G. DOS Version 2.10 is not limited to 16 block device units as previous versions were. The new limit is 63, but drives are assigned alphabetically through the collating sequence, so after drive Z, the drive "characters" get a little strange (like <, \, >).

Device Header

A device header is required at the beginning of a device driver. Here is what the Device Header looks like:

Description	Definition
Pointer to next device header	DWORD
Attribute	WORD
Pointer to device strategy	WORD
Pointer to device interrupt	WORD
Name/unit field	8 BYTES

Next Device Header Field

The pointer to the next device header field is a double word field (offset followed by segment) that is set by DOS at the time the device driver is loaded. However, it is important that this field be set to –1 prior to load time (when it is on the disk as a .COM file) unless there is more than one device driver in the .COM 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 next Device Header field set to –1.

Attribute Field

The next field in the header describes to the system the attributes of the device. They are as follows:

bit 15 = 1 if character device
0 if block device
bit 14 = 1 if IOCTL is supported
0 if it is not
bit 13 = 1 if non IBM format (block only)
0 if IBM format
bit 3 = 1 if current clock device
0 if it is not
bit 2 = 1 if current NUL device
0 if it is not
bit 1 = 1 if current standard output device
0 if it is not
bit 0 = 1 if current standard input device
0 if it is not

All other bits must be off.

The most important bit is bit 15, which tells the system that it is a block or a character device. With the exception of bits 13 and 14, the rest are for giving character devices special treatment and mean nothing on a block device. These *special treatment* bits allow you to tell DOS that your new device driver is the new standard input device and standard output device (the CON device). This can be done by setting bits 0 and 1 to 1. Similarly, a new CLOCK$ device could be installed by setting that attribute bit.

Although there is a NUL device attribute bit, the NUL device *cannot be reassigned.* This is an attribute that exists for DOS so it can tell if the NUL device is being used. The non IBM format bit applies only to block devices and affects the operation of the Get BPB (BIOS Parameter Block) device call (covered later in this chapter). The other bit of interest is the IOCTL bit. This is used for both block and character devices, and tells DOS whether the device is able to handle control strings (through the IOCTL system call).

If a driver cannot process control strings, it should initially set this bit to 0. This way DOS can return an error if an attempt is made through the IOCTL system call to send or receive control strings to the device. A device that is able to process such control strings should initialize this bit to 1. For devices of this type, DOS will make the calls to the IOCTL input and the IOCTL output device functions to send and receive IOCTL strings.

The IOCTL functions allow data to be sent to and from the device without actually doing a normal read or write. In this way, the device can use the data for its own use (like setting a baud rate, stop bits, changing form lengths, etc.). It is up to the device to interpret the information passed to it, but it must not be treated as a normal I/O request.

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.

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 DOS will fill in this location with the value returned by the driver's INIT code. (Refer to "Installation of Device Drivers" in this chapter.)

Creating a Device Driver

In order to create a device driver that DOS can install, a .COM file must be created with the Device Header at the start of the file. Remember 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 .COM 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.

DOS always processes installable device drivers before handling the default devices, so to install a new CON device, simply name the device CON (just be sure 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 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 at the same place every time.

Installation of Device Drivers

DOS Version 2.10 allows new device drivers to be installed dynamically at boot time by reading and processing the device options in the CONFIG.SYS file.

DOS calls a device driver at it's strategy entry point first, passing in a Request Header the information describing what DOS wants the device driver to do.

The strategy routine does not perform the request, but rather it enqueues the request (saves a pointer to the Request Header). The second entry point is the interrupt routine, and is called by DOS immediately after the strategy routine returns. The "interrupt" routine is called with no parameters. Its function is to perform the operation based on the queued request and set up any return information.

DOS passes the pointer to the Request Header in ES:BX. This structure consists of a fixed length header (Request Header) followed by data pertinent to the operation to be performed.

> Note: It is the responsibility of the device driver to preserve the machine state (for example, save all registers on entry, and restore them on exit).
>
> The stack used by DOS will have enough room on it to save all of the registers. If more stack space is needed, it is the device drivers responsibility to allocate and maintain another stack.
>
> All calls to device drivers are FAR calls, and FAR returns should be executed to return to DOS. (See "Sample Device Driver" listing at the end of this chapter.)

Request Header

BYTE length in bytes of the Request Header plus any data at the end of the Request Header
BYTE unit code The subunit the operation is for (minor device). Has no meaning for character devices.
BYTE command code
WORD Status
8 BYTE area reserved for DOS
Data appropriate to the operation

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.

Command Code

The command code field in the Request Header can have the following values:

Code	Function
0	INIT
1	MEDIA CHECK (Block only, NOP for character)
2	BUILD BPB (Block only, NOP for character)
3	IOCTL input (only called if IOCTL bit is 1)
4	INPUT (read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Character devices only)
6	INPUT STATUS (Character devices only)
7	INPUT FLUSH (Character devices only)
8	OUTPUT (write)
9	OUTPUT (write) with verify
10	OUTPUT STATUS (Character devices only)
11	OUTPUT FLUSH (Character devices only)
12	IOCTL output (only called if IOCTL bit is 1)

BUILD BPB and MEDIA CHECK

BUILD BPB and MEDIA CHECK, for block devices only, are explained here.

DOS calls MEDIA CHECK first for a drive unit. DOS passes it's current Media Descriptor byte (see "Media Descriptor Byte" later in this chapter). MEDIA CHECK returns one of the following four results:

- Media Not Changed
- Media Changed
- Not Sure

DOS will call BUILD BPB under the following two conditions:

- If "Media Changed" is returned
- If "Not Sure" is returned and there are no dirty buffers (buffers with changed data, not yet written to disk).

Status Word

The status word in the Request Header.

E R R	RESERVED	B U S	D O N	ERROR CODE (bit 15 on)

The status word is zero on entry and is set by the driver interrupt routine on return. Remember that this word is stored low byte first in memory.

Bit 8 is the done bit. When set it means the operation is complete. For DOS 2.10 the Driver just sets it to one when it exits.

Bit 15 is the error bit. If it is set, then the low 8 bits of the status word indicate the error. The errors are:

00 Write Protect Violation
01 Unknown Unit
02 Device Not Ready
03 Unknown command
04 CRC Error
05 Bad Drive Request Structure Length
06 Seek Error
07 Unknown Media
08 Sector Not Found
09 Printer Out of Paper
0A Write Fault
0B Read Fault
0C General Failure

Bit 9 is the busy bit that is set by status calls.

For output on character devices: If it 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 it 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 the user has typed something. DOS assumes all character devices have an input *type ahead* buffer. Devices that do not have them should always return busy = 0 so that 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 and never again. There are several things returned by the INIT routine. First, there is a location of the first free byte of memory after the device driver (like a terminate and stay resident) that is stored in the ending address field. In this manner, initialization code can be used once and thrown away in order to save space.

After setting the ending address field, a character device driver can set the status word and return. While block devices are installed in the same way as character devices, they must return additional information. The number of units for the device driver is returned, and this determines the logical names that the devices will have. For example, if the current maximum logical device letter is F at the time of the install call, and the block device driver INIT routine returns 3 units, then their logical names will be G, H, and I. This mapping is determined by the position of the driver in the device list, and the number of units on the device. The number of units returned by INIT will override the value in the name/unit field of the Device Header.

In addition, a pointer to a BPB (BIOS Parameter Block) pointer array is also returned. This is a pointer to an array of *n* word pointers, where *n* is the number of units defined. These word pointers point to BPBs. In this way, if all of the units are the same, the entire array can point to the same BPB in order to save space.

> **Note:** This array must be protected (below the free pointer set by the return).

The BPB (BIOS Parameter Block) contains information pertinent to the devices like sector size, sectors per allocation unit, etc. The sector size in the BPB cannot be greater than the maximum allowed (set at DOS initialization time).

The last thing that INIT of a block device must pass back is the "media descriptor byte". This byte means nothing to DOS, but is passed to devices so that they know what parameters DOS is currently using for a particular Drive-Unit.

Block devices may take several approaches; they may be *dumb* or *smart*. A dumb device would define a unit (and therefore a BPB) for each possible media drive combination. Unit 0 = drive 0 single side, unit 1 = drive 0 double side, etc. For this approach, media descriptor bytes would mean nothing. A smart device would allow 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 (sector size in BPB must be as large as maximum sector size that DOS is currently using). Smart drivers will use the "media byte" to pass information about what media is currently in a unit.

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 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.

INIT

Command code=0

ES:BX

13–BYTE Request Header
BYTE number of units (not set by character devices)
DWORD Ending Address
DWORD Pointer to BPB array (not set by character devices)

The driver must do the following:

- Set the number of units (block devices only).
- Set up the pointer to the BPB array (block devices only).
- Perform any initialization code (to modems, printers, etc.).
- Set up the ending address for resident code.
- Set the status word in the Request Header.

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 DOS uses. For the sake of simplicity, it is recommended that all of the device drivers in a single .COM file return the same ending address.

MEDIA CHECK

Command code=1

ES:BX

13–BYTE Request Header
BYTE Media Descriptor from DOS
BYTE return information

The driver must perform the following:

- Set the return byte:
  - –1 Media has been changed
  - 0 Don't know if media has been changed
  - 1 Media has not been changed
- Set the status word in the Request Header.

BUILD BPB (BIOS Parameter Block)

Command code=2

ES:BX

13-BYTE Request Header
BYTE Media Descriptor from DOS
DWORD Transfer Address (buffer address)
DWORD Pointer to BPB table

The driver must perform the following:

- Set the pointer to the BPB.
- Set the status word in the Request Header.

The driver must determine the correct media that is currently in the unit to return the pointer to the BPB table. The way the buffer is used (pointer passed by DOS) is determined by the non-IBM format bit in the attribute field of the device header. If the bit is zero (device is IBM format compatible) then the buffer contains the first sector of the FAT (most importantly the FAT id byte). The driver must not alter this buffer in this case. If the bit is a one, then the buffer is a one sector scratch area that can be used for anything.

If the device is IBM format compatible, then it must be true that the first sector of the first FAT is located at the same sector for all possible media. This is because the FAT sector is read *before* the media is actually determined.

The information relating to the BPB for a particular media is kept in the boot sector for the media. In particular, the format of the boot sector is:

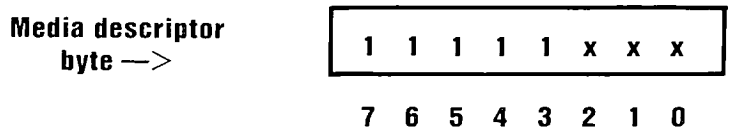
3 BYTE near JUMP to boot code
8 BYTE OEM name and version
WORD bytes per sector
BYTE sectors per allocation unit (must be a power of 2)
WORD reserved sectors (starting at logical sector 0)
BYTE number of FATs
WORD number of root dir entries (maximum allowed)

WORD number of sectors in logical image (total sectors in media, including boot sector, directories, etc.)
BYTE media descriptor
WORD number of sectors occupied by a single FAT
WORD sectors per track
WORD number of heads
WORD number of hidden sectors

The three words at the end are optional. DOS does not care about them because they are not part of the BPB. They are intended to help the device driver understand the media. Sectors per track may be redundant because it can be calculated from the total size of the disk. The number of heads is useful for supporting different multi-head drives that have the same storage capacity but a different number of surfaces. The number of hidden sectors is useful for supporting drive partitioning schemes.

MEDIA Descriptor Byte

Currently the media descriptor byte has been defined for a few media types:



Bit	Meaning	
0	1=2 sided	0=not 2 sided
1	1=8 sector	0=not 8 sector
2	1=removable	0=not removable
3–7	must be set to 1	

Examples of current DOS media descriptor bytes:

- 3 1/2" or 5 1/4" Diskettes:

  hex FC 1 sided 9 sector
  hex FD 2 sided 9 sector
  hex FE 1 sided 8 sector
  hex FF 2 sided 8 sector

- Fixed Disks:

  hex F8 (Fixed disk)

- 8" Diskettes:

  Hex FE (IBM 3740 Format). Single sided, single density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 1 reserved sector, 2 FATs, 68 directory entries, 77*26 sectors.

Hex FD (IBM 3740 Format). Dual sided, single density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 4 reserved sectors, 2 FATs, 68 directory entries, 77*26 sectors.

Hex FE. Single sided, double density, 1024 bytes per sector, soft sectored, 1 sector per allocation unit, 1 reserved sector, 2 FATs, 192 directory entries, 77*8*2 sectors.

Note: The two MEDIA descriptor bytes that are the same for 8" diskettes (hex FE) is not a misprint. To establish whether a diskette is single density or double density, a read of a single density address mark should be made. If an error occurs, the media is double density.

INPUT or OUTPUT

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

ES:BX

13–BYTE Request Header
BYTE Media descriptor byte
DWORD transfer address (buffer address)
WORD byte/sector Count
WORD starting sector number (no meaning on character devices)

The driver must perform the following:

- Do the requested function.
- Set the actual number of sectors (bytes) transferred.
- Set the status word in the Request Header.

  Note: No error checking is performed on an IOCTL call. However, the driver must set the return sector (byte) count to the correct number transferred.

The following applies to block device drivers:

Under certain circumstances the device driver may be asked to do a write operation of 64K bytes that seems to be a *wrap around* of the transfer address in the device driver request packet. This arises due to an optimization added to the write code in DOS. It will only happen on WRITEs that are within a sector size of 64K bytes on files that are being extended past the current end of file. It is allowable for the device driver 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 xxxx:1 could ignore the last two bytes.

Remember: A program that uses DOS function calls can never request an input or output operation of more than FFFFH bytes: therefore, a wrap around in the transfer (buffer) segment cannot occur. It is for this reason that you can ignore bytes that would have wrapped around in the transfer segment.

Non Destructive Input No Wait

Command code=5

ES:BX

13-BYTE Request Header
BYTE read from device

The driver must perform the following:

- Return a byte from the device.
- Set the status word in the Request Header.

This call is analagous to the console input status call on previous versions of DOS. 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 Input). This call allows DOS to look ahead one input character.

STATUS

Command codes=6 and 10

ES:BX

13-BYTE Request Header

All the driver must do is perform the operation and set the status word in the Request Header accordingly.

FLUSH

Command codes=7 and 11

ES:BX

13-BYTE Request Header

This call tells the driver to flush (terminate) all pending requests that it has knowledge of. Its primary use is to flush the input queue on character devices. The driver must set status word in the Request Header upon return.

The Time of Day Device Driver

The JX PC DOS 2.10 diskette contains two files which support the Time of Day (TOD) device drive. This device driver can be used when the 384KB/TOD or the 256KB/TOD RAM card is installed. See Appendix C of the DOS Reference Manual for more information.

The Memory Device Driver

The JX PC DOS 2.10 diskette contains four files which support the Memory Device Driver. This device driver can be used when the JX has 256KB or more RAM. See Appendix E of the DOS Reference Manual for more information.

Chapter 4. DOS Disk Allocation

Contents

DOS Disk Directory 4-4
DOS File Allocation Table 4-8
How to Use the File Allocation Table 4-10

All disks and diskettes formatted by DOS are created with a sector size of 512 bytes. The DOS area (entire diskette for diskettes, DOS partition for fixed disks) is formatted as follows:

Boot record – variable size
First copy of file allocation table – variable size
Second copy of file allocation table – variable size
Root directory – variable size
Data area

Allocation of space for a file (in the data area) is done only when needed (it is not pre-allocated). The space is allocated one cluster (unit of allocation) at a time. A cluster is always one or more consecutive sector numbers, and all of the clusters for a file are "chained" together in the File Allocation Table.

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 sector numbers on the lowest-numbered head, then all the sector numbers on the next head, and so on until all sectors on all heads of the track are used. Then, the next sector to be used will be sector 1 on head 0 of the next track.

For fixed disk, the size of the file allocation table and directory are determined when FORMAT initializes it, and are based on the size of the DOS partition.

For diskettes, 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

Files in the data area are not necessarily written sequentially on the disk. The data area space 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. (Refer to the description of the "DOS File Allocation Table".)

DOS Disk Directory

FORMAT initially builds the root directory for all disks. Its location (logical sector number) and the maximum number of entries are available through the device driver interfaces.

Since directories other than the root directory are actually files, there is no limit to the number of entries they may contain. Subdirectories can be read as data files, using an extended FCB with the appropriate attribute byte.

All directory entries are 32 bytes in length, and are in the following format (byte offsets are in decimal):

0-7 Filename. The first byte of this field indicates its status.

- hex 00 Never been used. This is used to limit the length of directory searches, for performance reasons.
- hex E5 Was used, but the file has been erased.
- hex 2E The entry is for a directory. If the second byte is also hex 2E, then the cluster field contains the cluster number of this directory's parent directory (hex 0000 if the parent directory is the root directory).

Any other character is the first character of a filename.

8-10 Filename extension.

11 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 output using function call hex 3D results in an error code being returned. This value can be used along with other values below.
- 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, and may exist only in the root directory.

10 The entry defines a subdirectory, and is excluded from normal directory searches.

20 Archive bit. The bit is set on whenever the file has been written to and closed. It is used by the BACKUP and RESTORE commands for determining whether or not the file was changed since it was last backed up. This bit can be used along with other attribute bits.

Note: The system files (IBMBIO.COM and IBMDOS.COM) 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 CHMOD function call.

12-21 Reserved.

22-23 Time the file was created or last updated. The time is mapped in the bits as follows:

```
<    hh    ><    mm    ><    xx    >
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

where:

hh is the binary number of hours (0-23)
mm is the binary number of minutes (0-59)
xx is the binary number of two-second increments

Note: The time is stored with the least significant byte first.

24-25 Date the file was created or last updated. The mm/dd/yy are mapped in the bits as follows:

<							25							> <							24							>
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
y	*y*	*y*	*y*	*y*	*y*	*y*	*m*	*m*	*m*	*m*	*d*	*d*	*d*	*d*	*d*

where:

mm is 1-12
dd is 1-31
yy is 0-119 (1980-2099)

Note: The date is stored with the least significant byte first.

26-27 Starting cluster; the cluster number of the first cluster in the file.

Note that the first cluster for data space on all fixed disks and diskettes is always cluster 002.

The cluster number is stored with the least significant byte first.

> **Note:** System programmers, see "DOS File Allocation Table" for details about converting cluster numbers to logical sector numbers.

28-31 File size in bytes. The first word contains the low-order part of the size. Both words are stored with the least significant byte first.

DOS File Allocation Table

This information is presented for the benefit of system programmers who wish to develop device drivers. It explains how 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. We wish to emphasize that this information should not be used for any other purpose. We recommend that system utilities use the DOS file management function calls rather than interpreting the FAT.

The File Allocation Table (FAT) is used by DOS to allocate disk space for a file, one cluster at a time.

The FAT consists of a 12-bit entry (1.5 bytes) for each cluster on the disk.

Note that the first two FAT entries map a portion of the directory; these FAT entries contain indicators of the size and format of the disk.

The second and third bytes always contain hex FFFF. The first byte is used as follows:

Hex Value	Meaning
FF	Dual sided, 8 sector-per-track diskette.
FE	Single sided, 8 sector-per-track diskette.
FD	Dual sided, 9 sector-per-track diskette.
FC	Single sided, 9 sector-per-track diskette.
F8	Fixed disk

The third FAT entry begins the mapping of the data area (cluster 002).

Each entry contains three hexadecimal characters, either:

000	if the cluster is unused and available, or
FF8-FFF	to indicate the last cluster of a file, or
XXX	any other hexadecimal 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.

> **Note:** The values FF0-FF7 are used to indicate reserved clusters (FF7 indicates a bad cluster if it is not part of an allocation chain), and FF8-FFF are used as end-of-file marks.

The File Allocation Table always occupies the sector or sectors immediately following the boot record. If larger than 1 sector, the sectors occupy consecutive numbers. Two copies of the FAT are written, one following the other, for integrity. The FAT is read into one of the DOS buffers whenever needed (open, allocate more space, etc.), and that buffer is given a high priority to keep it in memory as long as possible, for performance reasons.

How to Use the File Allocation Table

Obtain the *starting cluster* of the file from the directory entry.

Now, 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; otherwise, keep the high-order 12 bits.

5. If the resultant 12 bits are hex FF8-FFF, there are no more clusters in the file. 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 INT 25 and 26 and by DEBUG):

1. Subtract 2 from the cluster number.

2. Multiply the result by the number of sectors per cluster.

3. Add the logical sector number of the beginning of the data area.

Chapter 5. DOS Interrupts and Function Calls

Contents

Interrupts 5-3
Function Calls 5-13
    Error Return Table 5-14
Invoking DOS Functions 5-16

Interrupts

Note: We recommend that a program wishing to examine or set the contents of any interrupt vector use the DOS function calls (hex 35 and hex 25) provided for those purposes, and avoid referencing the interrupt vector locations directly.

DOS reserves interrupt types hex 20 to hex 3F for its use. This means absolute memory locations hex 80 to hex FF are reserved by DOS. The defined interrupts are as follows with all values in hexadecimal.

20 Program terminate. Issuing Interrupt hex 20 is the traditional way to exit from a program. This vector transfers to the logic in DOS for restoration of the terminate, Ctrl-Break, and critical error exit addreses to the values they had on entry to the program. All file buffers are flushed. All files changed in length should be closed (see function call hex 10 and hex 3E) prior to issuing this interrupt. If the changed file is not closed, its length, date, and time are not recorded correctly in the directory.

In order for a program to pass a completion (or error) code when terminating, it must use either function call hex 4C (exit) or hex 31 (terminate and stay resident). These two new methods are preferred over using interrupt hex 20, and the codes returned by them can be interrogated in batch processing (see ERRORLEVEL subcommand of batch processing).

Important: Every program must ensure that the CS register contains the segment address of its Program Segment Prefix control block prior to issuing interrupt hex 20.

21 Function request. Refer to "Function Calls" in this chapter.

22 Terminate address. The address found at this interrupt location is the address to which control transfers when the program terminates. This address is copied into the program's Program Segment Prefix at the time the segment is created. If a program wishes to execute a second program it must set the terminate address prior to executing the new program (unless EXEC is used). Otherwise, when the second program executes, its termination would cause transfer to its host's termination address. This address, as well as the Ctrl-Break address below, may be set via DOS function call hex 25. Do not issue this interrupt directly.

23 Ctrl-Break exit address. If the user enters Ctrl-Break during standard input, standard output, standard printer or RS-232C card operations, an interrupt type hex 23 is executed. (If BREAK is on, the interrupt hex 23 is issued on *any* function call.) If the Ctrl-Break routine saves all registers, it may end with a return-from-interrupt instruction (IRET) to continue program execution. If the program returns with a long return, the carry flag is used to determine whether the program will be aborted or not; if the carry flag is set, it will be aborted, otherwise execution will continue (as with a return by IRET). If the Ctrl-Break interrupts functions

9 or 10, buffered I/O, then ∧ C, carriage-return, and linefeed are output. If execution is then continued with an IRET, I/O continues from the start of the line. When the interrupt occurs, all registers are set to the value they had when the original function call to DOS was made. There are no restrictions on what the Ctrl-Break handler is allowed to do, including DOS function calls,as long as the registers are unchanged if IRET is used.

If the program creates a new segment and loads in a second program which itself changes the Ctrl-Break address, the termination of the second program and return to the first causes the Ctrl-Break address to be restored to the value it had before execution of the second program. (It is restored from the second program's Program Segment Prefix.)

24 Critical error handler vector. When a critical error occurs within DOS, control is transferred with an interrupt 24H. On entry to the error handler, AH will have its bit 7=0 (high-order bit) if the error was a disk error (probably the most common occurrence), bit 7=1 if not.

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

The registers will be set up for a retry operation, and an error code will be in the lower half of the DI register with the upper half undefined. These are the error codes:

Error Code	Description
0	Attempt to write on write-protected diskette
1	Unknown unit
2	Drive not ready
3	Unknown command
4	Data error (CRC)
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

DOS registers from issuing INT hex 24

AX
BX
CX
DX
SI
DI
BP
DS
ES

User registers at time of original INT hex 21 request

IP
CS
FLAGS

From the original interrupt hex 21 from the user to DOS

The registers are set such that if an IRET is executed, DOS will respond according to (AL) as follows:

(AL)=0 ignore the error.

=1 retry the operation.

=2 terminate the program through interrupt hex 23.

Disk Errors

If it is a hard error on disk (AH bit 7=0), register AL contains the failing drive number (0 = drive A, etc.); AH bits 0-2 indicate the affected disk area and whether it was a read or write operation, as follows:

Bit 0=0 if read operation,
1 if write operation.

Bits 2-1 (affected disk area)

0 0 DOS area (system files)
0 1 file allocation table
1 0 directory
1 1 data area

Other Errors

If AH bit 7=1, then the error occurred on a character device, or was the result of a bad memory image of the FAT. 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.

If a character device, the contents of AL are unpredictable, the error code is in DI as above.

Notes:

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

2. For disk errors, this exit is taken only for errors occurring during an interrupt hex 21 function call. It is not used for errors during an interrupt hex 25 or hex 26.

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 DOS function calls. If necessary, it may use calls 1 through 12. Use of any other call will destroy the DOS stack and will leave 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 itself rather than returning to DOS, it should restore the application program's registers from the stack, remove all but the last 3 words on the stack, then issue an IRET. This will return to the program immediately after the INT 21 that experienced the error. Note that if this is done, DOS will be in an unstable state until a function call higher than 12 is issued.

   The device header pointed to by BP:SI is formatted as follows:

DWORD Pointer to next device (FFFF if last device)
WORD Attributes Bit 15 = 1 if character device, 0 if block if bit 15 is 1 Bit 0 = 1 if Current standard input Bit 1 = 1 if Current standard output Bit 2 = 1 if Current NUL device Bit 3 = 1 if Current CLOCK device Bit 14 is the IOCTL bit
WORD Pointer to Device driver strategy entry point
WORD Pointer to Device driver interrupt entry point
8-BYTE character device named field for block devices the first byte is the number of units

To tell if the error occurred on a block or character device you must look at bit 15 in the attribute field (WORD at BP:SI+4).

If the name of the character device is desired, look at the eight bytes starting at BP:SI+10.

25 Absolute disk read. This transfers control directly to the DOS BIOS. Upon return, the original flags are still on the stack (put there by the INT instruction). This is necessary because return information is passed back in the current flags. Be sure to pop the stack to prevent uncontrolled growth. The request is as follows:

(AL)	Drive number (for example, 0=A or 1=B)
(CX)	Number of sectors to read
(DX)	Beginning logical sector number
(DS:BX)	Transfer address

The number of sectors specified are transferred between the given drive and the transfer address. *Logical sector numbers* are obtained by numbering each sector sequentially starting from track 0, head 0, sector 1 (logical sector 0) and continuing along the same head, then to the next head until the last sector on the last head of the track is counted. Thus, logical sector 1 is track 0, head 0, sector 2; logical sector 2 is track 0, head 0, sector 3; and so on. Numbering then continues with sector 1 on head 0 of the next track. Note that although the sectors are sequentially numbered (for example, sectors 2 and 3 on track 0 in the example above), they may not be physically adjacent on disk, due to interleaving.

All registers except the segment registers are destroyed by this call. If the transfer was successful the carry flag (CF) will be zero. If the transfer was not successful CF=1 and (AX) will indicate the error as follows. (AL) is the DOS error code that is the same as the error code returned in the low byte of DI when an INT hex 24 is issued, and (AH) will contain:

hex 80	Attachment failed to respond
hex 40	SEEK operation failed
hex 20	Controller failure
hex 10	Bad CRC on diskette read
hex 04	Requested sector not found
hex 03	Write attempt on write-protected diskette
hex 02	Address mark not found

26 Absolute disk write. This vector is the counterpart of interrupt 25 above. Except for the fact that this is a write, the description above applies.

27 Terminate but stay resident. This vector is used by programs that are to remain resident when COMMAND regains control. This is the traditional method for DOS programs to remain resident upon termination.

After initializing itself, the program must set DX to its last address plus one in the segment in which it is executing (the offset at which other programs can be loaded), then execute an INT 27H. DOS then considers the program as an extension of DOS, so the program is not overlaid when other programs are executed. This concept is very useful for loading programs such as user-written interrupt handlers that must remain resident.

Notes:

1. This interrupt must *not* be used by .EXE programs which are loaded into the high end of memory.

2. This interrupt restores the interrupt 22, 23, and 24 vectors in the same manner as INT 20. Therefore, it cannot be used to install permanently resident Ctrl-Break or Critical Error Handler routines.

3. The maximum size of memory that can be made resident by this method is 64K.

4. A new DOS function call has been established that allows the terminating program to pass a completion (or error) code to DOS, that can be interpreted within batch processing (see function call hex 31). This is the preferred method.

28	Used internally by DOS.
29-2E	Reserved for DOS.
2F	Used internally by DOS.
30-3F	Reserved for DOS.

Function Calls

DOS provides a wide variety of function calls for character device I/O, file management, memory management, date and time functions, execution of other programs, and others. They are grouped as follows (call numbers are in hexadecimal):

0-12	Traditional character device I/O
12-24	Traditional file management
25-26	Traditional non-device functions
27-29	Traditional file management
2A-2E	Traditional non-device functions
2F-38	Extended function group
39-3B	Directory group
3C-46	Extended file management group
47	Directory group
48-4B	Extended memory management group
4C-4F	Extended function group
54-57	Extended function group

Functions 2F through 57 are new for DOS Versions 2.00 and 2.10. Where similar functions exist in both this group and the group of traditional calls, we recommend using the new calls. They have been defined with simpler interfaces and provide more powerful functions than their traditional counterparts.

When DOS takes control, it switches to an internal stack. User registers are preserved unless information is passed back to the requester as indicated in the specific requests. The user stack needs to be sufficient to accommodate the interrupt system. It is recommended that it be hex 80 in addition to the user needs.

Error Return Table

Many of the new function calls return the carry flag clear if the operation was successful. If an error condition was encountered, the carry flag is set, and AX contains one of the following binary error return codes:

Code	Condition
1	Invalid function number
2	File not found
3	Path not found
4	Too many open files (no handles left)
5	Access denied
6	Invalid handle
7	Memory control blocks destroyed
8	Insufficient memory
9	Invalid memory block address
10	Invalid environment
11	Invalid format
12	Invalid access code
13	Invalid data
15	Invalid drive was specified
16	Attempted to remove the current directory
17	Not same device
18	No more files

Several of the calls accept an ASCIIZ string as input. This consists of an ASCII string containing an optional drive specifier, followed by a directory path, and in some cases a filename. The string is terminated by a byte of binary zeros. For example:

```
B:\LEVEL1\LEVEL2\FILE1
```

followed by a byte of zeros.

> Note: All calls that accept path names will accept a forward slash or a backslash as a path separator character.

The new calls supporting files or devices use an identifier known as a "handle." When you create or open a file or device with the new calls, a 16-bit binary value is returned in AX. This is the "handle" (sometimes known as a token) that you will use in referring to the file after it's been opened.

The following handles are pre-defined by DOS and can be used by your program. You do not need to open them before using them:

0000	Standard input device. Input can be redirected.
0001	Standard output device. Output can be redirected.
0002	Standard error output device. Output cannot be redirected.
0003	Standard auxiliary device.
0004	Standard printer device.

Invoking DOS Functions

Most of the 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 these ways:

1. Place the function number in AH and execute a long call to offset hex 50 in your Program Segment Prefix.

2. Place the function number in AH and issue interrupt type hex 21.

3. There is an additional mechanism provided for pre-existing 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 DOS function dispatcher. Register AX is always destroyed if this mechanism is used; otherwise, it is the same as normal function calls. This method is valid only for function calls 0-24 (hexadecimal).

The functions are as follows with all values in hexadecimal.

0 Program terminate. The terminate, Ctrl-Break, and critical error exit addresses are restored to the values they had on entry to the terminating program, from the values saved in the Program Segment Prefix. All file buffers are flushed, but any files which have been changed in length but not closed will not be recorded properly in the directory. Control transfers to the terminate address. This call performs exactly the same function as INT 20H. It is the program's responsibility to ensure that the CS register contains the segment address of its Program Segment Prefix control block prior to calling this function.

> Note: Calls hex 1 through hex C use the standard devices listed at the end of the "Error Return Table" in this chapter.

1 Keyboard input. Waits for a character to be read at the standard input device (unless one is ready), then echoes the character to the standard output device and returns it in AL. The character is checked for a Ctrl-Break. If Ctrl-Break is detected, an interrupt hex 23 is executed.

> **Note:** For functions 1, 6, 7, and 8, extended ASCII codes will require two function calls. (See the IBM Personal Computer JX *BASIC* manual for a description of the extended ASCII codes.) The first call returns 00 as an indicator that the next call will return an extended code.

2 Display output. The character in DL is output to the standard output device. The backspace character results in moving the cursor left one position, writing a space at this position and remaining there. If a Ctrl-Break is detected after the output, an interrupt hex 23 is executed.

3 Auxiliary (RS-232C card) input. Waits for a character from the standard auxiliary device, then returns that character in AL.

Notes:

1. Auxiliary (AUX, COM1, COM2) support is unbuffered and non-interrupt driven.

2. At startup, DOS initializes the first auxiliary port to 2400 baud, no parity, one stop bit, and 8-bit word.

3. The auxiliary function calls (3 and 4) do not return status or error codes. For greater control, it is recommended that the ROM BIOS routine (INT hex 14) be used.

4 Auxiliary (RS-232C card) output. The character in DL is output to the standard auxiliary device.

5 Printer output. The character in DL is output to the standard printer device.

6 Direct console I/O. If DL is hex FF, AL returns with the zero flag clear and an input character from the standard input device if one is ready. If a character is not ready, the zero flag will be set. If DL is not hex FF, then DL is assumed to have a valid character that is output to the standard output device. This function does not check for Ctrl-Break, or Ctrl-PrtSc.

7 Direct console input without echo. Waits for a character to be read at the standard input device (unless one is ready), then returns the character in AL. As with function 6, no checks are made on the character.

8 Console input without echo. This function is identical to function 1, except the key is not echoed.

9 Print string. On entry, DS:DX must point to a character string in memory terminated by a $ (hex 24). Each character in the string will be output to the standard output device in the same form as function 2.

A Buffered keyboard input. On entry, DS:DX point to an input buffer. The first byte must not be zero and specifies the number of characters the buffer can hold. Characters are read from the standard input device and placed in the buffer beginning at the third byte. Reading the standard input device and filling the buffer continues until Enter is read. If the buffer fills to one less than the maximum number of characters it can hold, then each additional character read is ignored and causes the bell to ring, until Enter is read. The second byte of the buffer is set to the number of characters received, excluding the carriage return (hex 0D), which is always the last character.

B Check standard input status. If a character is available from the standard input device, AL will be hex FF. Otherwise, AL will be 00. If a Ctrl-Break is detected, an interrupt type hex 23 is executed.

C Clear keyboard buffer and invoke a keyboard function. Clear the keyboard buffer of any pre-typed characters, then execute the function number in AL (only 1, 6, 7, 8, and A are allowed). This forces the system to wait until a character is typed.

D Disk reset. Flushes all file buffers. Files changed in size but not closed are not properly recorded in the disk directory. This function need not be called before a diskette change if all files written have been closed.

E Select disk. The drive specified in DL (0=A, 1=B, etc.) is selected (if valid) as the default drive. The number of drives (total of diskette and fixed disk drives) is returned in AL. If the system has only one diskette drive, it will be counted as two to be consistent with the philosophy of thinking of the system as having logical drives A and B. BIOS equipment determination (INT 11H) can be used as an alternative method, returning the actual number of physical diskette drives.

F Open file. On entry, DS:DX point to an unopened file control block (FCB). The current directory is searched for the named file and AL returns hex FF if it is not found. If it is found, AL returns 00 and the FCB is filled as follows:

If the drive code was 0 (default drive), it is changed to the actual drive used (1=A, 2=B, etc.). This allows changing the default drive without interfering with subsequent operations on this file. The current block field (FCB bytes C-D) is set to zero. The size of the record to be worked with (FCB bytes E-F) is set to the system default of hex 80. The size of the file and the date are set in the FCB from information obtained from the directory.

It is your responsibility to set the record size (FCB bytes E-F) to the size you wish to think of the file in terms of, if the default hex 80 is insufficient. It is also your responsibility to set the random record field and/or current record field. These actions should be done after open but before any disk operations are requested.

10 Close file. This function must be called after file writes to ensure all directory information is updated. On entry, DS:DX point to an opened FCB. The current disk directory is searched and if the file is found, its position is compared with that kept in the FCB. If the file is not found in its correct position in the current directory, it is assumed the diskette was changed and AL returns hex FF. Otherwise, the directory is updated to reflect the status in the FCB and AL returns 00.

11 Search for the first entry. On entry, DS:DX point to an unopened FCB. The current disk directory is searched for the first matching filename (name could have "?"s indicating any letter matches) and if none are found, AL returns hex FF. Otherwise, AL returns 00 and the locations at the disk transfer address are set as follows:

If the FCB provided for searching was an extended FCB, then the first byte at the disk transfer address is set to hex FF, followed by five bytes of zeros, then the attribute byte from the search FCB, then the drive number used (1=A, 2=B, etc.), then the 32 bytes of the directory entry. Thus, the disk transfer address contains a valid unopened extended FCB with the same search attributes as the search FCB.

If the FCB provided for searching was a normal FCB, then the first byte is set to the drive number used (1=A, 2=B), and the next 32 bytes contain the matching directory entry. Thus, the disk transfer address contains a valid unopened normal FCB.

Notes:

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

1. If the FCB attribute byte 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.

The attribute bits are defined in the "DOS Disk Directory" section of Chapter 4 of this manual.

12 Search for the next entry. After function 11 has been called and found a match, function 12 may be called to find the next match to an ambiguous request (?s in the search filename). Both inputs and outputs are the same as function 11. The reserved area of the FCB keeps information necessary for continuing the search, so no disk operations may be performed with this FCB between a previous function 11 or 12 call and this one.

13 Delete file. On entry, DS:DX point to an unopened FCB. All matching current directory entries are deleted. If no directory entries match, AL returns hex FF, otherwise AL returns 00.

14 Sequential read. On entry, DS:DX point to an opened FCB. The record addressed by the current block (FCB bytes C-D) and the current record (FCB byte 1F) is loaded at the disk transfer address, then the record address is incremented. (The length of the record is determined by the FCB record size field.) If end-of-file is encountered, AL returns either 01 or 03. A return of 01 indicates no data in the record; 03 indicates a partial record is read and filled out with zeros. A return of 02 means there was not enough space in the disk transfer segment to read one record, so the transfer was ended. AL returns 00 if the transfer was completed successfully.

15 Sequential write. On entry, DS:DX point to an opened FCB. The record addressed by the current block and current record fields (size determined by the FCB record size field) is written from the disk transfer address (or, in the case of records less than sector sizes, is buffered up for an eventual write when a sector's worth of data is accumulated). The record address is then incremented. If the diskette is full, AL returns 01. A return of 02 means there was not enough space in the disk transfer segment to write one record, so the transfer was ended. AL returns 00 if the transfer was completed successfully.

16 Create file. On entry, DS:DX point to an unopened FCB. The current disk directory is searched for a matching entry, and if found, it is re-used. If no match was found, the directory is searched for an empty entry, and AL returns FF if none is found. Otherwise, the entry is initialized to a zero-length file, the file is opened (see function F), and AL returns 00.

The file may be marked *hidden* during its creation by using an extended FCB containing the appropriate attribute byte.

17 Rename file. On entry, DS:DX point to a modified FCB which has a drive code and a file name in the usual position, and a second file name starting 6 bytes after the first (DS:DX+hex 11) in what is normally a reserved area. Every matching occurrence of the first name in the current directory is changed to the second (with the restriction that two files cannot have the same name and extension). If "?"s appear in the second name, then the corresponding positions in the original name will be unchanged. AL returns FF if no match was found or if an attempt was made to rename to a filename that already existed, otherwise 00.

18 Used internally by DOS.

19 Current disk. AL returns with the code of the current default drive (0=A, 1=B, etc.).

1A Set disk transfer address. The disk transfer address is set to DS:DX. DOS does not allow disk transfers to wrap around within the segment, or overflow into the next segment.

1B Allocation table information. On return, DS:BX point to a byte containing the FAT identification byte for the default drive, DX has the number of allocation units, AL has the number of sectors per allocation unit, and CX has the size of the physical sector.

1C Allocation table information for specific drive. This call is identical to call hex 1B except that on entry, DL contains the number of the drive from which the information should be gotten (0 = default, 1= A, etc.).

1D Used internally by DOS.

1E Used internally by DOS.

1F Used internally by DOS.

20 Used internally by DOS.

21 Random read. On entry, DS:DX point to an opened FCB. The current block and current record fields are set to agree with the random record field, then the record addressed by these fields is read into memory at the current disk transfer address. If end-of-file is encountered, AL returns either 01 or 03. If 01 is returned, no more data is available. If 03 is returned, a partial record is available filled out with zeros. A return of 02 means there was not enough space in the disk transfer segment to read one record, so the transfer was ended. AL returns 00 if the transfer was completed successfully.

22 Random write. On entry, DS:DX point to an opened FCB. The current block and current record fields are set to agree with the random record field, then the record addressed by these fields is written (or in the case of records not the same as sector sizes – buffered) from the disk transfer address. If the disk is full AL returns 01. A return of 02 means there was not enough space in the disk transfer segment to write one record; so, the transfer was ended. AL returns 00 if the transfer was completed successfully.

23 File size. On entry, DS:DX point to an unopened FCB. The current directory is searched for the first matching entry and if none is found. AL returns FF. Otherwise, the random record field is set to the number of records in the file (in terms of the record size field rounded up) and AL returns 00.

> **Note:** Be sure to set the FCB record size field before using this function call; otherwise, erroneous information will be returned.

24 Set random record field. On entry. DS:DX point to an opened FCB. This function sets the random record field to the same file address as the current block and record fields.

25 Set interrupt vector. The interrupt vector table for the interrupt type specified in AL is set to the 4-byte address contained in DS:DX. Note that the original contents of the interrupt vector can be obtained through call hex 35.

26 Create a new program segment. On entry, DX has a segment number at which to set up a new program segment. The entire hex 100 area at location zero in the current program segment is copied into location zero in the new program segment. The memory size information at location 6 in the new segment is updated and the current termination, Ctrl-Break exit and critical error addresses from interrupt vector table entries for interrupt types 22, 23, and 24 are saved in the new program segment starting at hex 0A. They are restored from this area when the program terminates.

> **Note:** Use of this call should be avoided, now that DOS contains the EXEC function call (hex 4B).

27 Random block read. On entry, DS:DX point to an opened FCB, and CX contains a record count that must not be zero. The specified number of records (in terms of the record size field) are read from the file address specified by the random record field into the disk transfer address. If end-of-file is reached before all records have been read, AL returns either 01 or 03. A return of 01 indicates end-of-file and the last record is complete. A return of 03 indicates the last record is a partial record. If wrap-around above address hex FFFF in the disk transfer segment would have occurred, as many records as possible are read and AL returns 02. If all records are read successfully, AL returns 00. In any case, CX returns with the actual number of records read, and the random record field and the current block/record fields are set to address the next record (the first record not read).

28 Random block write. Essentially the same as function 27 above, except for writing and a write-protect check. If there is insufficient space on the disk, AL returns 01 and no records are written. If CX is zero upon entry, no records are written, but the file is set to the length specified by the random record field, whether longer or shorter than the current file size. (Allocation units are released or allocated as appropriate.)

29 Parse filename. On entry, DS:SI point to a command line to parse, and ES:DI point to a portion of memory to be filled with an unopened FCB. The contents of AL are used to determine the action to take, as shown below:

```
      <ignored>
bit:  7  6  5  4  3  2  1  0
```

If bit 0 = 1, then leading separators are scanned off the command line at DS:SI. Otherwise, no scan-off of leading separators takes place.

If bit 1 = 1, then the drive ID byte in the result FCB will be set (changed) *only* if a drive was specified in the command line being parsed.

If bit 2 = 1, then the filename in the FCB will be changed only if the command line contains a filename.

If bit 3 = 1, then the filename extension in the FCB will be changed only if the command line contains a filename extension.

Filename separators include the following characters : . ; , = + plus TAB and SPACE. Filename terminators include all of these characters plus \, <, >, |, /, ", [,], and any control characters.

The command line is parsed for a filename of the form d:filename.ext, and if found, a corresponding unopened FCB is created at ES:DI. If no drive specifier is present, the default drive is assumed. If no extension is present, it is assumed to be all blanks. If the character * appears in the filename or extension, then it and all remaining characters in the name or extension are set to ?.

If either ? or * appears in the filename or extension, AL returns 01; if the drive specifier is invalid AL returns FF; otherwise 00.

DS:SI will return pointing to the first character after the filename and ES:DI will point to the first byte of the formatted FCB. If no valid filename is present, ES:DI+1 will contain a blank.

> **Note:** This call is not useful for command lines containing path names.

2A Get date. Returns date in CX:DX. CX has the year (1980-2099 in binary), DH has the month (1-Jan, 2-Feb, etc.) and DL has the day. If the time-of-day clock rolls over to the next day, the date is adjusted accordingly, taking into account the number of days in each month and leap years.

2B Set date. On entry, CX:DX must have a valid date in the same format as returned by function 2A, above. If the date is indeed valid and the set operation is successful, AL returns 00. If the date is not valid, AL returns FF.

2C Get time. Returns with time-of-day in CX:DX, and the binary day of the week (0=Sunday) in AL. Time is actually represented as four 8-bit binary quantities as follows. CH has the hours (0-23), CL has minutes (0-59), DH has seconds (0-59), DL has 1/100 seconds (0-99). This format is readily converted to a printable form yet can also be used for calculations, such as subtracting one time value from another.

2D Set time. On entry, CX:DX has time in the same format as returned by function 2C, above. If any component of the time is not valid, the set operation is aborted and AL returns FF. If the time is valid, AL returns 00.

2E Set/reset verify switch. On entry, DL must contain 0, and AL must contain 1 to turn verify on, or 0 to turn verify off. When on, DOS will perform a verify operation each time it performs a diskette write to assure proper data recording. Although disk recording errors are very rare, this function has been provided for those user applications in which you may wish to verify the proper recording of critical data. Note that the current setting of the verify switch can be obtained through call hex 54.

2F Get DTA. On return, ES:BX contains the current DTA transfer address.

30 Get DOS version number. On return, AL contains the major version number. AH contains the minor version number. On return, BX and CX are set to zero.

31 Terminate process and remain resident (KEEP process). On entry, AL contains a binary exit code. DX contains the memory size value in paragraphs. This function call terminates the current process and attempts to set the initial allocation block to the number of paragraphs in DX. It will not free up any other allocation blocks belonging to that process. The exit code passed in AL is retrievable by the parent through Wait (function call hex 4D) and can be tested through the ERRORLEVEL batch subcommands.

32 Used internally by DOS.

33 Ctrl-Break check. On entry, AL contains 00 to request the current state of Control-Break checking, 01 to set the state. If setting the state, DL must contain 00 for OFF or 01 for ON. DL returns the current state (00 = OFF, 01 = ON).

34 Used internally by DOS.

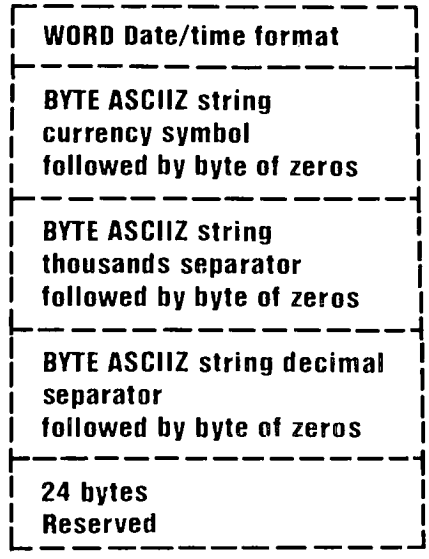
35 Get vector. On entry, AL contains a hexadecimal interrupt number. The CS:IP interrupt vector for the specified interrupt is returned in ES:BX. Note that interrupt vectors can be set through call hex 25.

36 Get disk free space. On entry, DL contains a drive: 0 = default, 1 = A, etc. On return, AX returns FFFF if the drive number was invalid. Otherwise, BX contains the number of available allocation units (clusters), DX contains the total number of clusters on the drive, CX contains the number of bytes per sector, and AX contains the number of sectors per cluster.

> Note: This call returns the same information in the same registers (except for the FAT pointer) as the get FAT pointer call (hex 1B) did in previous versions of DOS.

37 Used internally by DOS.

38 Return country dependent information (international). On entry, DS:DX points to a 32-byte block of memory in which returned information is passed and AL contains a function code. This function code must be zero. The following information is pertinent to international applications:



The date and time format has the following values and meaning:

0 = USA standard *h:m:s m/d/y*

1 = Europe standard *h:m:s d/m/y*

2 = Japan standard *h:m:s d:m:y*

39 Create a subdirectory (MKDIR). On entry, DS:DX contains the address of an ASCIIZ string with drive and directory path names. If any member of the directory path does not exist, then the directory path is not changed. On return, a new directory is created at the end of the specified path. Error returns are 3 and 5 (refer to error return table).

3A Remove a directory entry (RMDIR). On entry, DS:DX contains the address of an ASCIIZ string with the drive and directory path names. The specified directory is removed from the structure. The current directory cannot be removed. Error returns are 3 and 5 (refer to error return table). Note that code 5 is returned if the specified directory is not empty.

3B Change the current directory (CHDIR). On entry, DS:DX contains the address of an ASCIIZ string with drive and directory path names. If any member of the directory path does not exist, then the directory path is not changed. Otherwise, the current directory is set to the ASCIIZ string. Error return is 3 (refer to the error return table).

3C Create a file (CREAT). On entry, DS:DX contains the address of an ASCIIZ string with the drive, path, and filename. CX contains the attribute of the file. This function call 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 read/write access code. The file is opened for read/write, and the handle is returned in AX. Error returns are 3, 4, and 5 (refer to the error return table). If an error code of 5 is returned, either the directory was full or a file by the same name exists and is marked read-only. Note that the change mode function call (hex 43) can later be used to change the file's attribute.

3D Open a file. On entry, DS:DX contains the address of an ASCIIZ string with the drive, path, and filenames. AL contains the access code. On return, AX contains an error code or a 16-bit file handle associated with the file. The following access codes are allowed in AL:

AL = 0 – file is opened for reading.

AL = 1 – file is opened for writing.

AL = 2 – file is opened for both reading and writing.

The read/write pointer is set at the first byte of the file and the record size of the file is 1 byte (the read/write pointer can be changed through function call hex 42). The returned file handle must be used for subsequent input and output to the file. The file's date and time can be obtained or set through call hex 57, and its attribute can be obtained through call hex 43. Error returns are 2, 4, 5, and 12 (refer to the error return table).

> Note: This call will open any normal or hidden file whose name matches the name specified.

3E Close a file handle. On entry, BX contains the file handle that was returned by "open." On return, the file will be closed and all internal buffers are flushed. Error return is 6 (refer to the error return table).

3F Read from a file or device. On entry, BX contains the file handle. CX contains the number of bytes to read. DS:DX contains the buffer address. On return, AX contains the number of bytes read. If the value is zero, then the program has tried to read from the end of file. This function call transfers (CX) bytes from a file into a buffer location. It is not guaranteed that all bytes will be read. For example, reading from the keyboard will read at most one line of text. If this read is performed from the standard input device, the input can be redirected (see "Redirection of Standard Input and Output" in Chapter 1 of **Disk Operating System Reference** manual). Error returns are 5 and 6 (refer to the error return table).

40 Write to a file or device. On entry, BX contains the file handle. CX contains the number of bytes to write. DS:DX contains the address of the data to write. Write transfers (CX) bytes from a buffer into a file. AX returns the number of bytes actually written. If this value is not the same as the number requested, it should be considered an error (no error code is returned, but your program can compare these values). The usual reason for this is a full disk. If this write is performed to the standard output device, the output can be redirected (see "Redirection of Standard Input and Output" in Chapter 1 of the **Disk Operating System Reference** manual). Error returns are 5 and 6 (refer to the error return table).

41 Delete a file from a specified directory (Unlink). On entry, DS:DX contains the address of an ASCIIZ string with a drive, path, and filename. Global filename characters are not allowed in any part of the string. This function call removes a directory entry associated with a filename. Read-only files cannot be deleted by this call. To delete one of these files, you can first use call hex 43 to change the file's attribute to 0, then delete the file. Error returns are 2 and 5 (refer to the error return table).

42 Move file read/write pointer (Lseek). On entry, AL contains a method value. BX contains the file handle. CX:DX contains the desired offset in bytes (CX contains the most significant part). On return, DX:AX contains the new location of the pointer (DX contains the most significant part).

It moves the read/write pointer according to the following methods:

AL = 0 – The pointer is moved to offset (CX:DX) bytes from the beginning of the file.

AL = 1 – The pointer is moved to the current location plus offset.

AL = 2 – The pointer is moved to the end-of-file plus offset. This method can be used to determine file's size.

Error returns are 1 and 6 (refer to the error return table).

43 Change file mode (CHMOD). On entry, AL contains a function code, and DS:DX contains the address of an ASCIIZ string with the drive, path, and filename. If AL contains 01 then the file will be set to the attribute in CX. (See the "DOS Disk Directory" section of Chapter 4 for the attribute byte description.) If AL is 0 then the file's current attribute will be returned in CX. Error returns are 2, 3, and 5 (refer to the error return table).

44 I/O control for devices (IOCTL). On entry, AL contains the function value. BX contains the file handle. On return, AX contains the number of bytes transferred for functions 2, 3, 4, and 5 or status (00 = not ready, FF = ready) for functions 6 and 7, or an error code. Use IOCTL to Set or Get device information associated with open device handle, or send/receive control strings to the device handle. The following function values are allowed in AL:

AL = 0 – Get device information (returned in DX).

AL = 1 – Set device information (determined by DX). Currently, DH must be zero for this call.

AL = 2 – Read CX number of bytes into DS:DX from device control channel.

AL = 3 – Write CX number of bytes from DS:DX to device control channel.

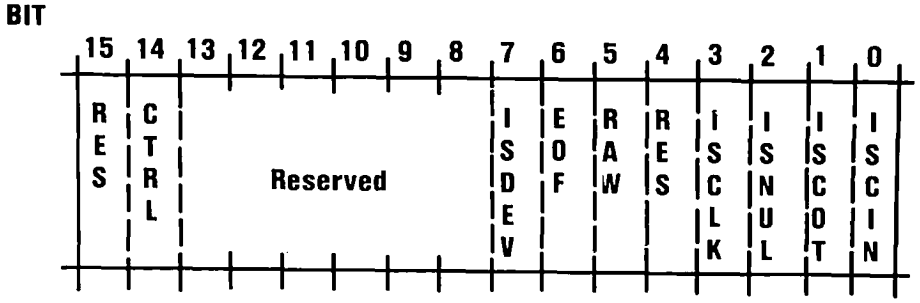
AL = 4 – Same as 2, but use drive number in BL (0 = default, 1 = A, etc.).

AL = 5 – Same as 3, but use drive number in BL (0 = default, 1 = A, etc.).

AL = 6 –Get input status.

AL = 7 – Get output status.

IOCTL can be used to get information about device channels. You can make calls on regular files, but only function values 0, 6, and 7 are defined in that case. All other calls return an "invalid function" error.



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.

BIN = 1 if operating in binary mode (no checks for Ctrl-Z).
= 0 if operating in ASCII mode (checking for Ctrl-Z as end-of-file).

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.

CTRL = 0 if this device cannot process 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 by function call hex 44.

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.

Note: DH must be zero for call AL=1.

Calls AL=2, AL=3, AL=4, AL=5. These four calls allow arbitrary control strings to be sent or received from a character device. The Call syntax is the same as the Read and Write calls, except for calls 4 and 5 which accept a drive number in BL instead of a handle in BX. An "invalid function" error is returned if the CTRL bit is zero. An "access-denied" code is returned by calls 4 and 5 if the drive is invalid. Error returns are 1, 6, and 13 (refer to the error return table).

Calls 6 and 7. These calls allow you to check if a file handle is ready for input or output. If used for a file, AL always returns FF until end-of-file is reached, then always returns 00 unless the current file position is changed through call hex 42. When used for a device, AL returns FF for ready or zero for not ready.

45 Duplicate a file handle (DUP). On entry, BX contains the file handle. On return, AX contains the returned file handle. This function call takes an opened file handle and returns a new file handle that refers to the same file at the same position. Error returns are 4 and 6 (refer to the error return table).

Note: If you move the read/write pointer of either handle, the pointer for the other handle will also be changed.

46 Force a duplicate of a handle (DUP). On entry, BX contains the file handle. CX contains a second file handle. On return, the CX file handle will refer to the same stream as the BX file handle. If the CX file handle was an open file, then it is closed first. Error return is 6 (refer to the error return table).

> **Note:** If you move the read/write pointer of either handle, the pointer for the other handle will also be changed.

47 Get Current directory. On entry, DL contains a drive number (0 = default, 1 = A, etc.) and DS:SI point to a 64-byte area of user memory. The full path name (starting from the root directory) of the current directory for the specified drive is placed in the area pointed to by DS:SI. Note that the drive letter will not be part of the returned string. The string will not begin with a backslash and will be terminated by a byte containing hex 00. The error returned is 15.

48 Allocate memory. On entry, BX contains the number of paragraphs requested. On return, AX:0 points to the allocated memory block. If the allocation fails, BX will return the size of the largest block of memory available in paragraphs. Error returns are 7 and 8 (refer to the error return table).

49 Free allocated memory. On entry, ES contains the segment of the block being returned. On return, a block of memory is returned to the system pool that was allocated by call hex 48. Error returns are 7 and 9 (refer to the error return table).

4A SETBLOCK-Modify allocated memory blocks. On entry, ES contains the segment of the block. BX contains the new requested block size in paragraphs. DOS will attempt to "grow" or "shrink" the specified block. If the call fails on a grow request, then on return, BX contains the maximum block size possible. Error returns are 7, 8, and 9 (refer to the error return table).

4B Load or execute a program (EXEC). This function call allows a program to load another program into memory and (default) begin execution of it. DS:DX points to the ASCIIZ string with drive, path, and filename of the file to be loaded. ES:BX points to a parameter block for the load and AL contains a function value. The following function values are allowed:

0 = Load and execute the program. A program segment prefix is established for the program and the terminate and control-break addresses are set to the instruction after the EXEC system call.

Note: When control is returned, all registers are changed including the stack. You must restore SS, SP and any other required registers before proceeding.

3 = Load, do not create the program segment prefix, and do not begin execution. This is useful in loading program overlays.

For each of these values, the block pointed to by ES:BX has the following format:

AL = 0 Load/execute program

WORD segment address of environment string to be passed
DWORD pointer to command line to be placed at PSP+ 80h
DWORD points to default FCB to be passed at PSP+ 5Ch
DWORD pointer to default FCB to be passed at PSP+ 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 newly created process after an EXEC. This is extremely powerful; the parent process has control over the meanings of standard input, output, auxiliary, and printer devices. 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 execute a sort program that takes its input from standard input and writes to standard output.

Also inherited (or copied from the parent) is an "environment." This is a block of text strings (less than 32K bytes total) that convey various configuration parameters. The following is the format of the environment (always on a paragraph boundary):

Byte ASCIIZ string 1
Byte ASCIIZ string 2
...
Byte ASCIIZ string n
Byte of zero

Typically the environment strings have the form:

parameter=value

For example, the string VERIFY=ON could be passed. A zero value of the environment address will cause the newly created process to inherit the parent's environment unchanged. The segment address of the environment is placed at offset hex 2C of the Program Segment Prefix for the program being invoked. Error returns are 1, 2, 5, 8, 10, and 11 (refer to the error return table).

Notes:

1. When your program received control, all of available memory was allocated to it. You must free some memory (see call hex 4A) before EXEC can load the program you are invoking. Normally, you would shrink down to the minimum amount of memory you need, and free the rest.

2. The EXEC call uses the loader portion of COMMAND.COM to perform the loading. If your program has overlaid the loader, this call will attempt to re-load the loader. If you have used the "Allocate Memory" call to allocate all of memory and the loader has been overlaid, the EXEC call will return an error due to insufficient memory to load the loader.

4C Terminate a process (Exit). On entry, AL contains a binary return code. This function call will terminate the current process, transferring control to the invoking process. In addition, a return code can be sent. The return code can be interrogated by the batch subcommands IF and ERRORLEVEL and by the wait function call (4D). All files opened by the call 3D are closed.

4D Retrieve the return code of a sub-process (Wait). This function call returns the Exit code specified by another process (via call hex 4C or call hex 31) in AX. It returns the Exit code only once. The low byte of this code is that sent by the exiting routine. The high byte is zero for normal termination, 01 if terminated by Crtl-Break, 02 if terminated as the result of a critical device error, or 03 if terminated by function call hex 31.

4E Find first matching file (FIND FIRST). On input, DS:DX points to an ASCIIZ string containing the drive, path, and filename of the file to be found. The filename portion can contain global filename characters. CX contains the attribute to be used in searching for the file. See function call hex 11 for a description of how the attribute bits are used for searches. If a file is found that matches the specified drive, path, and filename and attribute, the current DTA will be filled in as follows:

21 bytes – reserved for DOS use on subsequent find next calls

1 byte – attribute found

2 bytes – file's time

2 bytes – file's date

2 bytes – low word of file size

2 bytes – high word of file size

13 bytes – name and extension of file found, followed by a byte of zeros. All blanks are removed from the name and extension, and if an extension is present, it is preceded by a period. Thus, the name returned appears just as you would enter it as a command parameter. Such as, TREE.COM followed by a byte of zeros. Error returns are 2 and 18 (refer to the error return table).

4F Find next matching file. On input, the current DTA must contain the information that was filled in by a previous Find First call (hex 4E). No other input is required. This call will find the next directory entry matching the name that was specified on the previous Find First call. If a matching file is found, the current DTA will be set as described in call hex 4E. If no more matching files are found, error code 18 is returned (refer to the error return table).

50 Used internally by DOS.

51 Used internally by DOS.

52 Used internally by DOS.

53 Used internally by DOS.

54 Get verify state. On return, AL returns 00 if verify is OFF, 01 if verify is ON. Note that the verify switch can be set through call hex 2E.

55 Used internally by DOS.

56 Rename a file. On input, DS:DX points to an ASCIIZ string containing the drive, path, and filename of the file to be renamed. ES:DI points to an ASCIIZ string containing the path and filename to which the file is to be renamed. If a drive is used in this string, it must be the same as the drive specified or implied in the first string. The directory paths need not be the same, allowing a file to be moved to another directory and renamed in the process. Error returns are 3, 5, and 17 (refer to the error return table).

57 Get/Set a file's date and time. On input, AL contains 00 or 01. BX contains a file handle. If AL=00 on entry, DX and CX will return the date and time from the handle's internal table, respectively. If AL=01 on entry, the handle's date and time will be set to the date and time in DX and CX, respectively. The date and time formats are the same as those for the directory entry described in Chapter 4 of this manual, except that when passed in registers, the bytes are reversed (that is, DH contains the *low* order portion of the date, etc.). Error returns are 1 and 6 (refer to the error return table).

Chapter 6. DOS Control Blocks and Work Areas

Contents

DOS Memory Map 6-3
DOS Program Segment 6-5
Program Segment Prefix 6-9
File Control Block 6-11
    Standard File Control Block 6-12
    Extended File Control Block 6-15

DOS Memory Map

0000:0000	Interrupt vector table
0040:0000	ROM communication area
0050:0000	DOS communication area
XXXX:0000	IBMBIO.COM — DOS interface to ROM I/O routines
XXXX:0000	IBMDOS.COM — DOS interrupt handlers, service routines (INT 21 functions)
	DOS buffers, control areas, and installed device drivers
XXXX:0000	Resident portion of COMMAND.COM — Interrupt handlers for interrupts hex 22 (terminate), hex 23 (Ctrl-Break), hex 24 (critical error), and code to reload the transient portion.
XXXX:0000	External command or utility — (.COM or .EXE file)
XXXX:0000	User stack for .COM files (256 bytes)
XXXX:0000	Transient portion of COMMAND.COM — Command interpreter, internal commands, batch processor, external command loader.

Notes:

1. Memory map addresses are in segment:offset format. For example, 0070:0000 is absolute address hex 0700.

2. The DOS Communication Area is used as follows:

   0050:0000 Print screen status flag store

   - **0** Print screen not active or successful print screen operation
   - **1** Print screen in progress
   - **255** Error encountered during print screen operation

   0050:0001 Used by BASIC

   0050:0004 Single-drive mode status byte

   - **0** Diskette for drive A was last used
   - **1** Diskette for drive B was last used

   0050:0010 – 0021 Used by BASIC

   0050:0022 – 002F Used by DOS for diskette initialization

   0050:0030 – 0033 Used by MODE command

   All other locations within the 256 bytes beginning at 0050:0000 are reserved for DOS use.

3. User memory is allocated from the lowest end of available memory that will satisfy the request for memory.

DOS Program Segment

When you enter an external command, or invoke a program through the EXEC function call, DOS determines the lowest available address to use as the start of available memory for the program being invoked. This area is called the *Program Segment*.

At offset 0 within the Program Segment, DOS builds the Program Segment Prefix control block. (See below.) EXEC loads the program at offset hex 100 and gives it control.

The program returns from EXEC by a jump to offset 0 in the Program Segment Prefix, by issuing an INT 20, by issuing an INT 21 with register AH=0 or hex 4C, or by calling location hex 50 in the Program Segment Prefix with AH=0 or hex 4C.

> **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 call hex 4C.

All of these methods result in returning to the program that issued the EXEC. During this returning process interrupt vectors hex 22. hex 23, and hex 24 (terminate, Ctrl-Break, and critical error exit 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, control transfers to its transient portion. If a batch file was in process, it is continued; otherwise, COMMAND issues the system prompt and waits for the next command to be entered from the standard input device.

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 hex 2C 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 entire set of strings is terminated by another byte of zeros. The environment built by the command processor (and passed to all programs it invokes) will contain a COMSPEC= string at a minimum (the parameter on COMSPEC is the path used by DOS to locate the command processor on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings entered through the SET command (see Chapter 2 of the *DOS Reference* manual).

The environment that you are passed is actually a copy of the invoking process environment. If your application uses a "terminate and stay resident" concept, you should be aware that the copy of the environment passed to you is static. That is, your copy of the environment will not change even if subsequent SET, PATH, or PROMPT commands are issued.

- Offset hex 50 in the Program Segment Prefix contains code to invoke the DOS function dispatcher. Thus, by placing the desired function number in AH, a program can issue a long call to PSP+ 50 to invoke a DOS function, rather than issuing an interrupt type hex 21.

- Disk transfer address (DTA) is set to hex 80 (default DTA in the Program Segment Prefix).

- File control blocks at hex 5C and hex 6C are formatted from the first two parameters entered when the command was invoked. Note that if either parameter contained a path name, then the corresponding FCB will contain only a valid drive number. The filename field will not be valid.

- An Unformatted parameter area at hex 81 contains all the characters entered after the command name (including leading and imbedded delimiters), with hex 80 set to the number of characters. If the <,>, or | parameters were entered on the command line, they (and the filenames associated with them) will not appear in this area, because 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 reflects the validity of drive specifiers entered with the first two parameters 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)

For .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 the linker.

For .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 wishes to invoke another program through the EXEC function call, it must first free some memory through the Setblock (hex 4A) function call to provide space for the program being invoked.

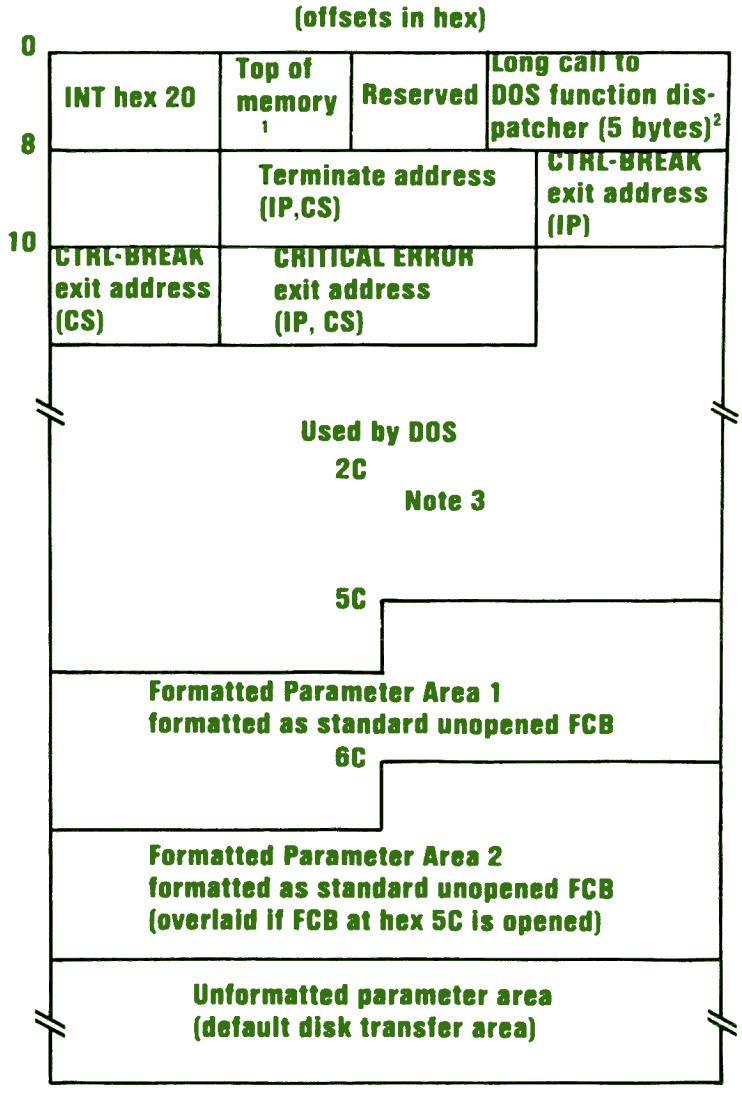
- The Instruction Pointer (IP) is set to hex 100.

- SP register is set to the end of the program's segment. The segment size at offset 6 is reduced by hex 100 to allow for a stack of that size.

- A word of zeros is placed on the top of the stack.

The Program Segment Prefix (with offsets in hexadecimal) is formatted as follows.

Program Segment Prefix



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.

4. Programs must not alter any part of the PSP below offset hex 5C.

File Control Block

Unshaded areas must be filled in by the using program.

Shaded areas are filled in by DOS and must not be modified.

Standard File Control Block

The standard file control block (FCB) is defined as follows, with the offsets in decimal:

Byte	Function
0	Drive number. For example, **Before open:** 0 – default drive 1 – drive A 2 – drive B etc. **After open:** 1 – drive A 2 – drive B etc. A 0 is replaced by the actual drive number during open.
1-8	Filename, left-justified with trailing blanks. If a reserved device name is placed here (such as LPT1), do not include the optional colon.
9-11	Filename extension, left-justified with trailing blanks (can be all blanks).
12-13	Current block number relative to the beginning of the file, starting with zero (set to zero by the open function call). A block consists of 128 records, each of the size specified in the logical record size field. The current block number is used with the current record field (below) for sequential reads and writes.

14-15 Logical record size in bytes. Set to hex 80 by the open function call. If this is not correct, you must set the value because DOS uses it to determine the proper locations in the file for all disk reads and writes.

16-19 File size in bytes. In this 2-word field, the first word is the low-order part of the size.

20-21 Date the file was created or last updated. The *mm/dd/yy* are mapped in the bits as follows:

<							21								>	<				20				>
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
y	*y*	*y*	*y*	*y*	*y*	*y*	*m*	*m*	*m*	*m*	*d*	*d*	*d*	*d*	*d*									

where:

mm is 1-12
dd is 1-31
yy is 0-119 (1980-2099)

22-31 Reserved for system use.

32 Current relative record number (0-127) within the current block. (See above.) You must set this field before doing *sequential* read/write operations to the diskette. (This field is not initialized by the open function call.)

33-36 Relative record number relative to the beginning of the file, starting with zero. You must set this field before doing *random* read/write operations to the diskette. (This field is not initialized by the open function call.)

If the record size is less than 64 bytes, both words are used. Otherwise, only the first three bytes are used. Note that if you use the File Control Block at hex 5C in the program segment, the last byte of the FCB overlaps the first byte of the unformatted parameter area.

Notes:

1. An unopened FCB consists of the FCB prefix (if used), drive number, and filename/extensions properly filled in. An open FCB is one in which the remaining fields have been filled in by the Create or Open function calls.

2. Bytes 0-15 and 32-36 must be set by the user program. Bytes 16-31 are set by DOS and must not be changed by user programs.

3. All word fields are stored with the least significant byte first. For example, a record length of 128 is stored as hex 80 at offset 14, and hex 00 at offset 15.

Extended File Control Block

The extended File Control Block is used to create or search for files in the disk directory that have special attributes.

It adds a 7-byte prefix to the FCB, formatted as follows:

Byte	Function
FCB-7	Flag byte containing hex FF to indicate an extended FCB.
FCB-6 to FCB-2	Reserved.
FCB-1	Attribute byte. See "DOS Disk Directory" in Chapter 4 of this manual for attribute bit definitions. Also refer to function call hex 11 (search first) for details on using the attribute bits during directory searches. This function is present to allow applications to define their own files as *hidden* (and thereby exclude them from directory searches), and to allow selective directory searches.

Any references in the DOS Function Calls (refer to Chapter 5 of this manual) to an FCB, whether opened or unopened, may use either a normal or extended FCB. If using an extended FCB, the appropriate register should be set to the first byte of the prefix, rather than the drive-number field.

Chapter 7. Executing Commands from Within an Application

With DOS Version 2.10, application programs may invoke a secondary copy of the command processor. Your program may pass a DOS command as a parameter and the secondary command processor will execute as though it had been entered from the standard input device. The procedure is:

1. Assure that adequate free memory (17K) exists to contain the second copy of the command processor and the command it is to execute.

2. Build a parameter string for the secondary command processor in the form:

   1 byte = length of parameter string
   xx byte = parameter string
   1 byte = hex 0D (carriage return)

   For example, the assembly statement below would build the string to cause execution of a DISKCOPY command:

   ```
   DB 14,"DISKCOPY A: B:",13
   ```

3. Use the EXEC function call (hex 4B, function value 0) to cause execution of the secondary copy of the command processor (the drive, directory and name of the command processor can be obtained from the COMSPEC=parameter in the environment passed to you at PSP+hex 2C). Remember to set offset 2 of the EXEC control block to point to the parameter string built above.

Chapter 8. Fixed Disk Information

Contents

Fixed Disk Architecture 8-3
System Initialization 8-4
Boot Record/Partition Table 8-6
Technical Information 8-8

The IBM Personal Computer Fixed Disk Support Architecture has been designed to meet the following objectives:

- Allow multiple operating systems to utilize the fixed disk without the need to dump/restore when changing operating systems.

- Allow a user-selected operating system to be started from the fixed disk.

Fixed Disk Architecture

The architecture is defined as follows:

- In order to *share* the fixed disk among operating systems, the disk may be logically divided into 1 to 4 "partitions." The space within a given partition is contiguous, and can be dedicated to a specific operating system. Each operating system may "own" only one partition. The number and sizes of the partitions are user-selectable through a fixed disk utility program (HDMGR). The partition information is kept in a partition table that is imbedded in the master fixed disk boot record on the first sector of the disk.

- Any operating system must consider its partition to be an entire disk, and must ensure that its functions and utilities do not access other partitions on the disk.

- Each partition can contain a boot record on its first sector, and any other programs or data that you choose–including a copy of an operating system. For example, the DOS FORMAT command may be used to format (and place a copy of DOS in) the DOS partition in the same manner that a diskette is formatted. You may designate a partition as "bootable" (active)—the master fixed disk boot record will cause that partition's boot record to receive control when the system is started or restarted.

System Initialization

The System initialization (or system boot) sequence is as follows:

1. System initialization first attempts to load an operating system from diskette drive A. If the drive is not ready or a read error occurs, it then attempts to read a master fixed disk boot record from the first sector of the first fixed disk on the system. If unsuccessful, or if no fixed disk is present, it invokes ROM BASIC.

2. If successful, the master fixed disk boot record is given control and it examines the partition table imbedded within it. If one of the entries indicates a "bootable" (active) partition, its boot record is read (from the partition's first sector) and given control.

3. If none of the partitions is bootable, ROM BASIC is invoked.

4. If any of the boot indicators are invalid, or if more than one indicator is marked as bootable, the message Invalid partition table is displayed and the system enters an enabled loop. You may then insert a system diskette in drive A and use system reset to restart from diskette.

5. If the partition's boot record can't be successfully read due to read errors, the message **Error loading operating system** appears and the system enters an enabled loop.

6. If the partition's boot record does not contain a valid "signature" (see "Boot Record/Partition Table"), the message Missing operating system appears, and the system enters an enabled loop.

   Note: When changing the size or location of any partition, you must ensure that all existing data on the disk has been backed up (the partitioning process will "lose track" of the previous partition boundaries).

Boot Record/Partition Table

A fixed disk boot record must be written on the first sector of all fixed disks, and contain:

1. Code to load (and give control to) the boot record for 1 of 4 possible operating systems.

2. A partition table at the end of the boot record. Each table entry is 16 bytes long, and contains the starting and ending cylinder, sector and head for each of 4 possible partitions, as well as the number of sectors preceding the partition and the number of sectors occupied by the partition. The "boot indicator" byte is used by the boot record to determine if one of the partitions contains a loadable operating system. Initialization utilities mark a user-selected partition as "bootable" by placing a value of hex 80 in the corresponding partition's boot indicator (setting all other partitions' indicators to zero at the same time.) The presence of the hex 80 tells the standard boot routine to load the sector whose location is contained in the following 3 bytes. That sector will be the actual boot record for the selected operating system, and it will be responsible for the remainder of the system's loading process (as it is from diskette). All boot records are loaded at absolute address 0:7C00.

The partition table (with its offsets into the boot record) is as follows:

Offs	Purpose	Head	Sector	Cylinder
13E	Reserved			
146	Reserved			
14E	Reserved			
156	Reserved			
15E	Partition 1 name			
166	Partition 2 name			
16E	Partition 3 name			
176	Partition 4 name			
17E	Reserved			
186	Reserved			
18E	Reserved			
196	Reserved			
19E	Reserved			
1A6	Reserved			
1AE	Reserved			
1B6	Reserved			
1BE	Partition 1 begin	boot ind	H	S	CYL
1C2	Partition 1 end	syst ind	H	S	CYL
1C6	Partition 1 rel sect	Low word		High word	
1CA	Partition 1 #sects	Low word		High word	
1CE	Partition 2 begin	boot ind	H	S	CYL
1D2	Partition 2 end	syst ind	H	S	CYL
1D6	Partition 2 rel sect	Low word		High word	
1DA	Partition 2 # sects	Low word		High word	
1DE	Partition 3 begin	boot ind	H	S	CYL
1E2	Partition 3 end	syst ind	H	S	CYL
1E6	Partition 3 rel sect	Low word		High word	
1EA	Partition 3 # sects	Low word		High word	
1EE	Partition 4 begin	boot ind	H	S	CYL
1F2	Partition 4 end	syst ind	H	S	CYL
1F6	Partition 4 rel sects	Low word		High word	
1FA	Partition 4 # sect	Low word		High word	
1FE	Signature	hex 55	hex AA		

Technical Information

When shipped by IBM, the 10-megabyte fixed disks are formatted with 512-byte sectors at an interleave factor of 8 (17 sectors per track, 4 heads per cylinder). They contain no data or boot records.

The boot indicator byte must contain 0 for a non-bootable partition, or hex 80 for a bootable partition. Only one partition can be marked bootable.

The "syst ind" field contains an indicator of the operating system that "owns" the partition. Each operating system can "own" only one partition.

The system indicators are:

hex 00 – unknown (unspecified)
hex 01 – DOS

Note: When you install another partition, bit 4 of "syst ind" must be 0.

The 1-byte fields labelled "CYL" contain the low-order 8 bits of the cylinder number–the high order 2 bits are in the high order 2 bits of the "S" (sector) field. This corresponds with ROM BIOS Interrupt hex 13 (disk I/O) requirements, to allow for a 10-bit cylinder number.

The fields are ordered in such a manner that only 2 MOV instructions are required to properly set up the DX and CX registers for a ROM BIOS call to load the appropriate boot record (fixed disk booting is only possible from the first fixed disk on a system, whose BIOS drive number (hex 80) corresponds to the boot indicator byte.)

All partitions are allocated in cylinder multiples and begin on sector 1, head 0. EXCEPTION: the partition which is allocated at the beginning of the disk starts at sector 2, to account for the disk's master boot record.

The number of sectors preceding each partition on the disk is kept in the 4-byte field labelled "rel sect." This value is obtained by counting the sectors beginning with cylinder 0, sector 1, head 0 of the disk, and incrementing the sector, head and then track values up to the beginning of the partition. Thus, if the disk has 17 sectors per track and 4 heads, and the second partition begins at cylinder 1, sector 1, head 0, the partition's starting relative sector is 68 (decimal)–there were 17 sectors on each of 4 heads on 1 track allocated ahead of it. The field is stored with the least significant word first.

The number of sectors allocated to the partition is kept in the"# of sects" field. This is a 4-byte field stored least significant word first.

The last 2 bytes of the boot record (hex 55AA) are used as a signature to identify a valid boot record. Both this record and the partition boot records are required to contain the signature at offset hex 1FE.

The Master disk boot record will invoke ROM BASIC if no indicator byte reflects a "bootable" system.

When a partition's boot record is given control, it is passed its partition table entry address in the DS:SI registers.

System programmers designing a utility to initialize/manage a fixed disk must provide the following functions at a minimum:

1. Write the master disk boot record/partition table to the disk's first sector to initialize it.

2. Perform partitioning of the disk—that is, create or update partition table information (all fields for the partition including the name field. Note that the name field for some partitions is related to that partition table. See the table on page 8-7.) when the user wishes to create a partition. This may be limited to creating a partition for only one type of operating system, but must allow repartitioning the entire disk, or adding a partition without interfering with existing partitions (user's choice).

3. Provide a means for marking a user-specified partition as bootable, and resetting the bootable indicator bytes for all other partitions at the same time.

Chapter 9. EXE File Structure and Loading

The .EXE files produced by the Linker program consist of two parts:

- Control and relocation information
- The load module itself

The control and relocation information, which is described below, is at the beginning of the file in an area known as the *header*. The load module immediately follows the header. The load module is the memory image of the module constructed by the linker.

The header is formatted as follows:

Hex Offset	Contents
00-01	Hex 4D, hex 5A—This is the LINK program's *signature* to mark the file as a valid .EXE file.
02-03	Length of image mod 512 (remainder after dividing the load module image size by 512).
04-05	Size of the file in 512-byte increments (*pages*), including the header.
06-07	Number of relocation table items that follow the formatted portion of the header.
08-09	Size of the header in 16-byte increments (*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.

Hex Offset	Contents
0C-0D	Maximum number of 16-byte paragraphs required above the end of the loaded program.
0E-0F	Offset of stack segment in load module (in segment form).
10-11	Value to be in the SP register when the module is given control.
12-13	Word checksum – negative sum of all the words in the file, ignoring overflow.
14-15	Value to be in the IP register when the module is given control.
16-17	Offset of code segment within load module (in segment form).
18-19	Offset of the first relocation item within the file.
1A-1B	Overlay number (0 for resident part of the program).

The relocation table follows the formatted area just described. The relocation table is made up of a variable number of relocation items. The number of items is contained at offset 06-07. The relocation item contains two fields—a 2-byte offset value, followed by a 2-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. This process is called *relocation* and is accomplished as follows:

1. A Program Segment Prefix is built following the resident portion of the program that is performing the load operation.

2. The formatted part of the header is read into memory (it's size is at offset 08-09).

3. The load module size is determined 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. Note that all files created by pre-release 1.10 LINK programs *always* placed a value of 4 at that location, regardless of actual program size. Therefore, we recommend that this field be ignored if it contains a value of 4. 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*.

4. The load module is read into memory beginning at the start segment.

5. The relocation table items are read into a work area (one or more at a time).

6. Each relocation table item segment value is added to the start segment value. This calculated segment, in conjunction with 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.

7. Once all relocation items have been processed, the SS and SP registers are set from the values in the header and 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 used to give the module control.

Chapter 10. DOS Memory Management

DOS keeps track of allocated and available memory blocks, and provides 3 function calls for application programs to communicate their memory needs to DOS. These calls are x'48' to allocate a memory block, x'49' to free a previously allocated memory block, and x'4A' (SETBLOCK) to change the size of an allocated memory block.

Memory is managed as follows:

DOS builds a control block for each block of memory, whether free or allocated. For example, if a program issues an 'allocate' call, DOS locates a block of free memory that satisfies the request, and will 'carve' the requested memory out of that block. The requesting program is passed the location of the first byte of the block that was allocated for it - a memory management control block, describing the allocated block, has been built for the allocated block (and a second memory management control block describes the amount of space left in the original free block of memory). When a SETBLOCK is done to shrink an allocated block, DOS builds a memory management control block for the area being freed, and adds it to the chain of control blocks. Thus, any program that changes memory that is not allocated to it, stands a good chance of destroying a DOS memory management control block. This causes unpredictable results that don't show up until an activity is performed where DOS uses its chain of control blocks (the normal result is a memory allocation error, for which the only cure is to restart the system).

When a program (command, application program) is to be loaded, DOS uses the EXEC function call (x'48') to perform the loading. This is the same function call that is available to application programs for loading other programs. This function call has 2 options - function 0, to load and execute a program (this is what the command processor uses to load and execute external commands), and function 3, to load an overlay (program) without executing it. Although both functions perform their loading in the same way (relocation is performed for EXE files), their handling of memory management is different.

For function 0 to load and execute a program, EXEC first allocates the largest available block of memory (the new program's PSP will be at offset 0 in that memory block). Then EXEC loads the program. Thus, in most cases, the new program 'owns' all of the memory from its PSP to the highest end of memory, including the memory occupied by the transient part of COMMAND.COM, which contains the loader. If the program were to issue its own EXEC function call to load and execute another program, the request would fail because no available memory exists to load the new program into. To further complicate things, if the calling program has overlaid the loader in COMMAND.COM, available memory has to be found to load the loader into, before the loader can load the required program.

> Note: for .EXE programs, the amount of memory allocated is the size of the program's memory image plus the value in the MAX ALLOC field of the file's header (offset x'0C'), if that much memory is available. If not, EXEC will allocate the size of the program's memory image plus the value in the MIN ALLOC field in the header (offset x'0A'). These fields are set by the LINKer. They are set to default values that cause the largest available memory block to be allocated.

A well-behaved program uses the SETBLOCK function call when it receives control, to shrink its allocated memory block down to the size it really needs (a .COM program should remember to set up its own stack before doing the SETBLOCK, since it is likely that the default stack supplied by DOS lies in the area of memory being freed). This frees unneeded memory, which can then be used for loading subsequent programs (and the loader, if necessary). This also benefits a multi-tasking environment, if that should become a reality in the future.

If the program requires additional memory during processing, it can obtain the memory via the allocate function call and later free it via the free memory call.

When a program loaded via function 0 exits, its initial allocation block (the block beginning with its PSP) is automatically freed before the calling program regains control. However, any other blocks allocated by the exiting program are *not* automatically freed - it is the responsibility of all programs to free any memory they allocate, prior to exiting to the calling program.

For function 3, to load an overlay, no PSP is built, and EXEC assumes the calling program has already allocated memory to load the new program into - it will *not* allocate memory for it. Thus, the calling program should either allow for the loading of overlays when it determines the amount of memory to keep when issuing the SETBLOCK call, or should initially free as much memory as possible. The calling program should then allocate a block (based on the size of the program to be loaded) to hold the program which will be loaded via the 'load overlay' call. Note that 'load overlay' does not check to see if the calling program actually own is the memory block it has been instructed to load into - it assumes the calling program has followed the rules. If the calling program does not own the memory into which the overlay is being loaded, there is a better-than-even chance that the program being loaded will overlay one of the control blocks that DOS uses to keep track of memory blocks.

Programs loaded via function 3 should *not* issue any SETBLOCK calls, since they don't own the memory they are operating in (the memory is own by the *calling* program).

Because programs loaded via function 3 are given control directly by (and return control directly to) the calling program with no DOS intervention, no memory is automatically freed when the called program exits - it is up to the calling program to determine the disposition of the memory that had been occupied by the exiting program. Note that if the exiting program had itself allocated any memory, it is responsible for freeing that memory before exiting.

Index

Special Characters

application, executing
commands within your 7-3
/S option 4-7

A

absolute disk read 5-10
absolute disk write 5-11
access, random 1-6
address terminate
interrupt 5-4
AH register 5-11
allocating disk space 1-6
allocating space 1-6
allocating diskette
space 4-8
allocation table
information 5-25
allocation, diskette 4-3
allocation 4-3, 4-4
disk allocation 4-3
architecture, 8088 1-3
structure 1-3
ASCII codes, extended 5-17
ASCIIZ string 5-15
Asynchronous
Communications
Adapter 5-18
attribute byte 6-15
attribute field 3-6
attribute, file 4-5
AUTOEXEC file 1-5
Auxiliary Asynchronous
Communications
Adapter 5-18
auxiliary input 5-18
auxiliary output 5-18
available functions, DOS 1-6
AX register 5-6, 5-16

B

batch file processor 1-5
BIOS 5-10
BIOS interface module 1-3
BIOS Parameter Block 3-15
block devices 3-5
block number, current 6-12
block read, random 5-26, 5-28
block write, random 5-26,
5-28
size 5-25
blocking/deblocking, data 1-3
BP register 5-6
BPB, what is 3-16
buffered standard input 5-19
buffers, file 5-3
built-in functions 1-3
BX register 5-6
byte, attribute 6-15
byte, flag 6-15

C

calls, function 5-13
chaining file sectors 1-6
character devices 3-5
check keyboard status 5-20
checksum methodology 1-5
CL register 5-16
close file 5-21
cluster number, relative 4-7
cluster, calculate 4-10
cluster, locate next 4-9, 4-10
cluster, starting 4-7
clusters 1-6, 4-3
  directory 4-5
  diskette directory 4-4
codes, error 5-6
COM filename extension 1-5
  .COM 1-5
  .EXE 1-5
COM programs 6-8
command processor 1-4
command processor, resident portion of 1-4
COMMAND.COM 1-4, 4-7, 6-3
Communications Adapter, Auxiliary Asynchronous 5-18
console I/O, direct 5-19
control blocks 6-15
control screen cursor 2-3
create file 5-24
creating a device driver 3-8
critical error handler vector 5-5
CS register 5-3, 5-7,.6-8
CTRL-BREAK exit address 1-5, 5-4
CTRL-BREAK handler 1-5
current block number 6-12
current disk 5-25
current relative record number 6-13
cursor control 2-3
CX register 5-6

D

data blocking/deblocking 1-3
date file created or updated 6-12
deblocking/blocking, data 1-3
  Initialization 1-4
delete file 5-23
device driver, creating 3-9
device drivers 3-3
device drivers, installation 3-10
device field, next 3-6
device header 3-6
devices, types of 3-4
DI register 5-6
direct console I/O 5-19
disk
  current 5-25
  reset 5-20
  select 5-20
disk error handling 1-5
disk errors 5-7
disk read, absolute 5-10
disk transfer address 6-7
disk transfer address, set 5-25
Disk Transfer Area (DIA) 1-7
disk write, absolute 5-11
display output 5-18
DOS environment 6-6
DS register 5-6, 6-8
DTA (Disk Transfer Area) 1-7
DX register 5-6

E

end-of-file mark 4-9
entries, search for 5-22
environment, DOS 6-6
error codes 5-6
error handling
  critical 1-5
  disk 1-5
error return table 5-14
error trapping 1-8
ES register 5-6, 6-8
EXE file structure 9-2
EXE filename extension 1-5, 6-3
  high 1-5
EXE files, load 9-2
EXE programs 6-8
executing commands within an application 7-2
extended ASCII codes 5-17
extended file control block 6-15
external commands 1-6

F

FAT (see File Allocation Table)
  allocating space 4-8
FCB 6-12
FCB (see File Control Block)
field name 3-8
field, attribute 3-7
File Allocation Table (FAT) 1-4, 1-6, 4-8
file allocation table, how to use 4-10
file buffers 5-3
File Control Block (FCB) 1-7, 6-11
  extended 6-15
  standard 6-11
File Management 1-6
file sectors
  chaining 1-6
  mapping 1-6
file size 6-13
file structure, .EXE 9-3
filename, parse 5-28
Fixed Disk Information 8-1
flag byte 6-15
FORMAT command 4-4
  hidden 4-4
format, device drivers 3-4
function call parameters 3-16
function calls 5-13
functions, available DOS 1-5
functions, built-in 1-3

G

get
  date 5-30
  time 5-30

H

header 9-3
hidden files 4-5, 5-24, 6-13
  attribute 4-6
high memory 1-5
  available functions 1-6
high/low loader switch 9-5

I

IBMBIO.COM 1-3, 1-4, 4-7
IBMDOS.COM 1-3, 4-7
initialization, DOS 1-3
input, auxiliary 5-18
installation of device drivers 3-10
instruction set, 8088 1-2
INT hex 24 1-8
INT 21 1-6
  random 1-6
  sectors 1-6
  sequential 1-6
interface module, BIOS 1-3
internal command processors 1-5
interrupt hex 20 5-3
interrupt hex 22 1-5, 5-4
interrupt hex 23 1-5, 5-4
interrupt hex 24 5-5
interrupt hex 25 5-10
interrupt hex 26 5-11
interrupt hex 27 5-11
interrupt mechanism, 8088 1-2
interrupt routines 3-8
interrupt vectors 1-4
interrupt, set 5-27
interrupts 5-3
IP register 5-7, 6-8
IRET 5-5

K

keyboard input 5-17
  buffered 5-19
keyboard status, check 5-20
keys, reassigning 2-3

L

linefeed 5-5
loading .EXE files 9-2
locate next cluster 4-9, 4-10
logical record size 6-12
logical sector numbers 5-10

M

mapping file sectors 1-6
memory management 10-1
memory map 6-3
MOV instruction 4-10

N

name field 3-8
next device field 3-6

O

open file 5-21
output, auxiliary 5-18
output, display 5-18

P

parameters, function call 3-16
parse filename 5-28
pointer to next device 3-9
print string 5-19
printer output 5-18
program segment
  create new 5-27
  DOS 6-5
Program Segment Prefix 1-4, 1-7, 6-8, 6-9
program terminate 5-17

R

random access 1-6
random block read 1-6, 5-28
random block write 1-6, 5-28
random read 5-26
random record field, set 5-24
random write 5-26
Read-Only Memory (ROM) 1-3
read, random 5-26
read, random block 5-28
read, sequential 5-24
reassign keys 2-2
record number, relative 6-13
record size, logical 6-11
relative record number 6-13
relocation 9-5
rename file 5-25
reset, disk 5-20
reset, system 1-4
resident portion of command processor 1-5
ROM (Read-Only Memory) 1-3
ROM BIOS routine 5-20
routines
  memory management 1-6
  ROM BIOS 5-19
routines, strategy/interrupt 3-8

S

screen cursor control 2-3
search for entries 5-22
sector numbers, logical 5-10
sectors, file 1-6
segment address 6-6
segment, create new program 5-27
segment, start 9-5
select disk 5-20
separators, filename 5-28
sequential read 5-24
sequential write 5-24
set
  date 5-30
  interrupt 5-27
  random record field 5-27
  time 5-30
  verify switch 5-30
set disk transfer address 5-25
SI register 5-6
single-drive system 5-20
size, file 6-13
SP register 6-8
space allocation 1-6, 4-3
SS register 6-8
stack, user 5-6
start segment 9-5
starting cluster 4-7
strategy routines 3-8
structure, DOS 1-3
switch, high/low loader 9-5
System Initialization 8-4
system prompt 1-5

T

technical information,
DOS 1-3
terminate address
interrupt 5-4
terminate program 5-17
terminate program
interrupt 5-4
terminators, filename 5-28
transfer address, disk 6-7
transient portion of command
processor 1-5

U

user stack 5-6

V

verify switch 5-30

W

work areas 6-3
wrap around 3-23
write, random 5-25
write, random block 5-28
write, sequential 5-24

Programming Series 110123-0 Printed in Japan

JX PC DOS Version 2.10 Technical Reference

IBM Americas/Far East Corporation