Disk Operating System Version 3.30

Technical Reference

Programming Family

80X0945

Disk Operating System Version 3.30

Technical Reference

Programming Family

First Edition (April 1987)

About This Book

Information in this book applies to DOS versions 2.10 to 3.30 unless specified in each chapter under the heading "Version Specific Information."

How This Book is Organized

This Technical Reference has two sections. Section 1 contains information about DOS versions 2.10 to 3.30 . Section 2 contains three chapters and one appendix, which cover information about the DOS utility programs that are on the DOS Utilities diskettes. The programs that are on the Technical Reference Utilities diskettes are DEBUG, LINK, EXE2BIN and the Library Manager from BASIC Compiler Version 2.00.

This book is written for the experienced DOS user, system programmer, and application developer. The information assumes that the reader is familiar with the 8088 architecture.

What comes in the DOS 3.30 package?

The DOS 3.30 package contains the following items:

- DOS 3.30 Technical Reference
- DOS 3.30 Technical Reference Quick Reference card
- One 5 1/4-inch DOS 3.30 Utilities diskette
- One 3 1/2-inch DOS 3.30 Utilities diskette

Contents

Section 1

Chapter 1. DOS Technical Information

Introduction

This chapter tells you about:

- DOS structure
- DOS initialization
- DOS functions
- Disk transfer area
- Error trapping

Version Specific Information

The following information in this chapter is specific to a version of DOS:

The Command Processor: For DOS 2.10, the transient portion of the command processor contains the EXEC routine that loads and executes external commands. For DOS versions 3.00 to 3.30, the resident portion of the command processor contains the EXEC routine.

DOS Structure

DOS consists of four components:

- The Boot Record
- The Read Only Memory BIOS Interface (IBMBIO.COM)
- The DOS Program File (IBMDOS.COM)
- The Command Processor (COMMAND.COM)

The Boot Record

The boot record begins on track 0, sector 1, side 0 of every diskette formatted by the DOS FORMAT command. The boot record is placed on diskettes to produce an error message if you try to start up the system with a nonsystem diskette in drive A. For fixed disks, the boot record resides on the first sector of the DOS partition. All media supported by DOS use one sector for the boot record.

Read Only Memory (ROM) BIOS Interface

The file IBMBIO.COM is the interface module to the Read Only Memory (ROM) BIOS. IBM Basic Input output System IBMBIO.COM provides a low-level interface to the ROM BIOS device routines.

The DOS Program File

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

When a user program calls these function routines, they accept high-level information by way of register and control block contents. For device operations, the functions translate the requirement into one or more calls to IBMBIO.COM to complete the request.

The Command Processor

The command processor, COMMAND.COM, consists of these parts:

1. A *resident* portion resides in memory immediately following IBMDOS.COM and its data area. This portion contains routines to process interrupts 22H (Terminate Address), 23H (Ctrl-Break Handler), and 24H (Critical Error Handling), as well as a routine to reload the transient portion if needed. For DOS 3.00 to 3.30, this portion also contains a routine to load and execute external commands, such as files with extensions of .COM or .EXE.

   Note: When a program terminates, a checksum methodology determines if the program has caused the transient portion to be overlaid. If the transient portion is overlaid, it is reloaded.

   All standard DOS error handling is done within this portion of COMMAND.COM. This includes displaying error messages and interpreting the replies of Abort, Retry, Ignore or

Fail. See the message "Disk error reading drive x" in Appendix A of the *DOS Reference*.

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

3. A *transient* portion is loaded at the high end of memory. This is the command processor itself, containing all of the internal command processors and the batch file processor. For DOS 2.10, this portion also contains a routine to load and execute external commands, such as files with extensions of .COM or .EXE.

   This portion of COMMAND.COM also produces the DOS prompt (such as **A>**), reads the command from the keyboard (or batch file), and executes the command. 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.

DOS Initialization

The system is initialized either by a system reset or by a power on. ROM BIOS first looks for the boot record on drive A. If the boot record is not found, ROM BIOS searches the active partition of the fixed disk. If it is not found there, ROM BIOS calls ROM BASIC. The following actions occur after a system initialization:

1. The boot record is read into memory and given control.

2. The boot record then checks the root directory to assure that the first two files are IBMBIO.COM and IBMDOS.COM. These two files must be the first two files, and they must be in that order.

3. The boot record loads IBMBIO.COM into memory.

4. The initialization code in IBMBIO.COM loads IBMDOS.COM, determines equipment status, resets the disk system, initializes the attached devices, loads the installable device drivers, sets the low-numbered interrupt vectors, relocates IBMDOS.COM downward, and calls the first byte of DOS.

5. DOS initializes its internal working tables, initializes the interrupt vectors for interrupts 20H through 27H, and builds a Program Segment Prefix for COMMAND.COM at the lowest available segment. For DOS versions 3.10 to 3.30, DOS initializes interrupt vectors for interrupts 0FH through 3FH.

6. IBMBIO.COM uses the EXEC function call to load and start the top-level command processor. The default command processor is COMMAND.COM.

Available DOS Functions

DOS provides a significant number of functions to user programs, all available through issuance of a set of interrupt and function calls. 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.

DOS provides two types of function calls that can be used for file management functions. They are:

- File control block (FCB) function calls
- Extended (Handle) function calls

See Chapter 4, "File Management Notes" for a description of FCB and Handle function calls. See Chapter 6, "DOS Interrupts and Function Calls" for detailed information on each individual call.

The Disk Transfer Area (DTA)

DOS uses an area in memory to contain the data for all file reads and writes that are performed with FCB function calls. This area in memory is called the *disk transfer area.* The disk transfer area (DTA) can also be called a *buffer.* This area can be at any location within the data area of your application program and should be set by your program.

Only one DTA can be in effect at a time, so your program must tell DOS what memory location to use *before* using any disk read or write functions. Use function call 1AH (Set Disk Transfer Address) to set the disk transfer address. Use function call 2FH (Get Disk Transfer Address) to get the disk transfer address. Refer to Chapter 6, "DOS Interrupts and Function Calls," for more information on these function calls. Once set, DOS continues to use that area for all disk operations until another function call 1AH is issued to define a new DTA. When a program is given control by COMMAND.COM, a default DTA large enough to hold 128 bytes is established at 80H into the program's Program Segment Prefix.

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 errors occur, DOS executes an interrupt 24H (Critical Error Handler Vector), 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 interrupt 24H vector to point to the new error handler. DOS provides error information by using the registers, and provides Abort, Retry, Ignore or Fail support by using return codes. See "Error Return Information" in Chapter 6, "DOS Interrupts and Function Calls," for more information on error codes.

Chapter 2. Installable Device Drivers

Introduction

This chapter tells you how to:

- Format a device driver
- Create a device driver
- Install a device driver

This chapter also provides information on the types of device drivers, the request header, and the CLOCK$ device.

The DOS device interface links the device drivers together in a chain. This allows you to add new device drivers for optional devices to DOS.

Version Specific Information

The following information in this chapter is specific to a version of DOS:

Attribute Field: Bit 6 (Get/Set Logical Device, Generic IOCTL) is for use with DOS versions 3.20 and 3.30. Bit 11 (removable media) is for use with DOS versions 3.00 to 3.30.

Command Code Field: Command code field values 13, 14, and 15 are for use with DOS versions 3.00 to 3.30. Command code 19 is only for use with DOS versions 3.20 and 3.30.

Status Word Field: Error codes 0DH, 0EH, and 0FH are only returned when using DOS versions 3.00 to 3.30.

Device Driver Functions:

- DOS versions 3.00 to 3.30 support removable media.
- The Media Check device driver function returns "Error" as a possibility if you are using DOS versions 3.00 to 3.30. Also for DOS 3.00 to 3.30, Media Check returns a DWORD pointer to the volume ID if a disk change has occurred.
- Media descriptor byte F9H for 5 1/4 inch, 15 sector media is supported by DOS versions 3.00 to 3.30.
- For DOS 3.00 to 3.30, the Input or Output device driver function returns a DWORD pointer to the volume identification if an invalid disk change has occurred.
- The Open or Close device driver function is for use with DOS versions 3.00 to 3.30.
- The Removable Media device driver function is for use with DOS versions 3.00 to 3.30.

Device Driver Format

A device driver is a memory image file or an .EXE file that contains all of the code needed to implement the device. It has a special header at the front of it that identifies the file as a device driver, defines the strategy and interrupt entry points, and defines various attributes of the device.

> **Note:** For device drivers, the memory image file must not use the ORG 100H. Because it does not use the program segment prefix, the device driver is simply loaded. Therefore, the memory image file must have an origin of 0 (ORG 0 or no ORG statement).

Types of Devices

There are two basic types of devices:

- Character devices
- Block devices

Character Devices

Character devices 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. Because character devices have only one name, they can support only one device.

Block Devices

Block devices are the "fixed disk or diskette drives" on the system. They can do random I/O in pieces called blocks, which are 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* by using the drive letters A, B, C, and so forth. 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, the first block device driver can be responsible for drives A, B, C, and D. This means that it has four units defined and therefore takes up four drive letters. The position of the driver in the chain of all drivers determines the way the drive units and drive letters correspond. For example, if the device driver is the first block driver in the device chain, and it defines four units, then those units are A, B, C, and D. If the second block driver defines three units, then those units are E, F,

and G. The limit is 26 devices with the letters A through Z assigned to the drives.

Device Header

A device driver requires a device header at the beginning of the file. Here is what the device header contains:

Field	Length
Pointer to next header	DWORD
Attribute	WORD
Pointer to device strategy routine	WORD
Pointer to device interrupt routine	WORD
Name/unit field	8 BYTES

Pointer to Next Device Header Field

The device header field is a pointer to the device header of the next device driver. It is a double-word field that is set by DOS at the time the device driver is loaded. The first word is an offset and the second word is the segment.

If you are loading only one device driver, set the device header field to -1 before loading the device. If you are loading more than one device driver, set the first word of this device header field to the offset of the next device driver's header. Set the device header field of the last device driver to -1.

Attribute Field

The attribute field is a word field that describes the attributes of the device driver to the system. The attributes are:

bit 15 = 1 character device
0 block device

bit 14 = 1 supports IOCTL
0 doesn't support IOCTL

bit 13 For block device drivers:
= 1 non-IBM format
0 IBM format
For character device drivers:
= 1 supports output-until-busy
0 doesn't support output-until-busy

bit 11 = 1 supports removable media
0 doesn't support removable media

bits 10-7 = 0 these bits must be off because they are reserved by DOS

bit 6 = 1 supports Get/Set Logical Device
0 doesn't support Get/Set Logical Device

Also

= 1 if the device supports Generic IOCTL function calls

= 0 if the device doesn't support Generic IOCTL function calls

bits 5-4 = 0 these bits must be off because they are reserved by DOS

bit 3 = 1 current clock device
0 not current clock device

bit 2 = 1 current NUL device
0 not current NUL device

bit 1 = 1 current standard output device
0 not current standard output device

bit 0 For character device drivers:
= 1 current standard input device
0 not current standard input device

Bit 15

Bit 15 is the device type bit. Use bit 15 to tell the system if the device driver is a block or character device. For block device drivers, bit 15 is 0. For charcter device drivers bit 15 is 1.

Bit 14

Bit 14 is the IOCTL bit. It is used for both character and block devices. Use bit 14 to tell DOS whether the device driver can handle control strings through the IOCTL function call (44H).

> **Note:** Bit 14 affects only IOCTL calls AL = 2 through AL = 5.

If a device driver cannot process control strings, it should set bit 14 to 0. This way DOS can return an error if an attempt is made through the IOCTL function call to send or receive control strings to the device. If a device can process control strings, it should set bit 14 to 1. This way, DOS makes the calls to the IOCTL input and output device function to send and receive IOCTL strings.

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

Bit 13

For block device drivers, bit 13 indicates the method the driver uses to determine the media type. For character device drivers, bit 13 indicates whether or not the driver supports output-until-busy.

If a block device driver uses information in the BPB to determine the media type, bit 13 should be set to 1. If the device driver uses the media descriptor byte to determine the media type, bit 13 should be 0.

For character device drivers, if output-until-busy is supported, bit 13 should be set to 1. If output-until-busy is not supported, bit 13 should be 0.

Output-until-busy is used by printer device drivers. Output-until-busy means that the device driver will send characters to the device if the device is ready. If the device driver supports output-until-busy and the device is not ready, the device driver will immediately return an error.

Bit 11

Bit 11 is the open/close removable media bit. Use bit 11 to tell DOS if the device driver can handle removable media.

Bit 6

Bit 6 is the Generic IOCTL bit for both character and block device drivers. If this bit is set, the device driver supports Generic IOCTL function calls

Bit 3

Bit 3 is the clock device bit. It is used for character devices only. Use bit 3 to tell DOS if your character device driver is the new CLOCK$ device.

Bit 2

Bit 2 is the NUL attribute bit. It is used for character devices only. Use bit 2 to tell DOS if your character device driver is a NUL device. Although there is a NUL device attribute bit, you cannot reassign the NUL device. This is an attribute that exists for DOS so that DOS can tell if the NUL device is being used.

Bits 0 and 1

For character devices, bits 0 and 1 are the standard input/standard output bits. Use these bits to tell DOS if your character device driver is the new standard input or standard output device.

Pointer to 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/Unit Field

This is an 8-byte field that contains the name of a character device or the unit of a block device. For character devices, the name is left-justified and the space is filled to 8 bytes. For block devices, the number of units can be placed in the first byte. This is optional because DOS fills in this location with the value returned by the driver's INIT code.

Creating a Device Driver

To create a device driver that DOS can install, perform the following:

- Create a memory image file or an .EXE file with a device header at the start of the file.
- Originate the code (including the device header) at 0, not at 100H.
- Set the next device header field. Refer to "Pointer to Next Device Header Field" for more information.
- Set the attribute field of the device header. Refer to "Attribute Field" for more information.
- Set the entry points for the interrupt and strategy routines.
- Fill in the name/unit field with the name of the character device, or the unit number of the block device.

DOS always processes installable character device drivers before handling the default devices. So to install a new CON device, simply name the device CON. Be sure to set the standard input device and standard output device bits in the attribute field 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.

Device Drivers

DOS installs new device drivers dynamically at boot time by reading and processing the DEVICE command in the CONFIG.SYS file. For example, if you have written a device driver called DRIVER1, to install it put this command in the CONFIG.SYS file:

```
device=driver1
```

DOS calls a device driver at its 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 queues the request or 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 has enough room on it to save all of the registers. If more stack space is needed, it is the device driver's responsibility to allocate and maintain another stack.

All calls to device drivers are FAR calls. FAR returns should be executed to return to DOS.

Installing Character Devices

One of the functions defined for each device is INIT. This routine is called only once when the device is installed and never again. The INIT routine returns the following:

- A location to the first free byte of memory after the device driver, like a terminate and stay resident that is stored in the ending address field. This way, the initialization code can be used once and thrown away to save space.

- After setting the ending address field, a character device driver can set the status word and return.

Installing Block Devices

Block devices are installed in the same way character devices are. The difference is that block devices return additional information. Block devices must also return:

- The number of units for the block device. This number 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 three logical units, the logical names of the devices are G, H, and I. The 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 overrides the value in the name/unit field of the device header.

- A pointer to a BPB (BIOS parameter block) pointer array. This is a pointer to an array of *n* word pointers where *n* is the number of units defined. These word pointers point to BPB's. This way, if all of the units are the same, the

entire array can point to the same BPB to save space.

The BPB contains information pertinent to the devices such as the sector size, the number of sectors per allocation unit, and so forth. The sector size in the BPB cannot be greater than the maximum allotted size set at DOS initialization time.

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

- The media descriptor byte. This byte is passed to devices so that they know what parameters DOS is currently using for a particular drive unit.

Request Header

The request header passes the information describing what DOS wants the device driver to do.

Field	Length
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.	BYTE
Status.	WORD
Area reserved for DOS.	8 – BYTE
Data appropriate to the operation.	Variable

Unit Code Field

The unit code field identifies which unit in a block device driver the request is for. For example, if a block device driver has three units defined, then the possible values of the unit code field would be 0, 1, and 2.

Command Code Field

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 control string (only called if IOCTL bit is 1)
4	INPUT (read)
5	NONDESTRUCTIVE 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 control string (only called if IOCTL bit is 1)
13	DEVICE OPEN (only called if OPEN/CLOSE/RM bit is set)
14	DEVICE CLOSE (only called if OPEN/CLOSE/RM bit is set)
15	REMOVABLE MEDIA (only called if OPEN/CLOSE/RM bit is set and device type is block)
19	GENERIC IOCTL REQUEST
23	GET LOGICAL DEVICE
24	SET LOGICAL DEVICE

Note: Command codes 13, 14, and 15 are for use with DOS versions 3.00 to 3.30. Command codes 19, 23, and 24 are only for use with DOS versions 3.20 and 3.30.

Status Field

The status field in the request header contains:

15	14-10	9	8	7-0
ERROR	RESERVED	BUSY	DONE	ERROR CODE (bit 15 on)

The status word field is zero on entry and is set by the driver interrupt routine on return.

Bit 15 is the error bit. If this bit is set, the low 8 bits of the status word (7-0) indicate the error code.

Bits 14 - 10 are reserved.

Bit 9 is the busy bit. It is only set by status calls and the removable media call. See "STATUS" and "REMOVABLE MEDIA" in this chapter for more information about the calls.

Bit 8 is the done bit. If it is set, it means the operation is complete. The driver sets the done bit to 1 when it exits.

Bits 7 – 0 are the low 8 bits of the status word. If bit 15 is set, bits 7 – 0 contain the error code. The error codes and errors are:

Error Codes	Description
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
0D	Reserved
0E	Reserved
0F	Invalid disk change

Device Driver Functions

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.

The following function call parameters are described:

- INIT
- MEDIA CHECK
- BUILD BPB (BIOS Parameter Block)
- MEDIA DESCRIPTOR BYTE
- INPUT or OUTPUT
- NONDESTRUCTIVE INPUT NO WAIT
- STATUS
- FLUSH
- OPEN or CLOSE
- REMOVABLE MEDIA
- GENERIC IOCTL REQUEST
- GET LOGICAL DEVICE
- SET LOGICAL DEVICE

INIT

Command code = 0

ES:BX

Field	Length
Request header	13 – BYTE
Number of units (not set by character devices)	BYTE
Ending address of resident program code	DWORD
Pointer to BPB array (not set by character devices) /pointer to remainder of arguments	DWORD
For DOS versions 3.10 to 3.30, this field contains the drive number	BYTE

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 the ending address of the resident program code.
- Set the status word in the request header.

To obtain information passed from CONFIG.SYS to a device driver at INIT time, the BPB pointer field points to a buffer containing the information passed in CONFIG.SYS following the =. This string may end with either a carriage return (0DH) or a linefeed (0AH). This information is read-only. Only system calls 01H – 0CH and 30H can be issued by the INIT code of the driver.

The last byte parameter contains the drive letter for the first unit of a block driver. For example, 0 = A, 1 = B etc.

If an INIT routine determines that it cannot set up the device and wants to abort without using any memory, follow this procedure.

- Set the number of units to 0.
- Set the ending address offset to 0.
- Set the ending address segment to the code segment (CS).

  Note: If there are multiple device drivers in a single memory image file, the ending address returned by the last INIT called is the one DOS uses. It is recommended that all device drivers in a single memory image file return the same ending address.

MEDIA CHECK

Command code = 1

ES:BX

Field	Length
Request header	13 – BYTE
Media descriptor from DOS	BYTE
Return	BYTE
If you are using DOS 3.00 to 3.30, this call returns a pointer to the previous volume ID (if bit 11 = 1 and disk change is returned)	DWORD

When the command code field is 1, DOS calls MEDIA CHECK for a drive unit and passes its current Media Descriptor byte. See "Media Descriptor Byte" later in this chapter for more information about the byte. MEDIA CHECK returns one of the following:

- Media not changed
- Media changed
- Not sure
- Error code

The driver must perform the following:

- Set the status word in the request header.
- Set the return byte:

  -1 Media has been changed

  0 Don't know if media has been changed

  1 Media has not been changed

The following method is used by the driver to determine how to set the Return byte.

- If the media is a fixed disk (non-removable media), set the return byte to **Media has not been changed**
- If 2 seconds have not passed since last successful access, set the return byte to **Media has not been changed**
- If changeline not available, set the return byte to **Don't know if media has been changed**
- If changeline is available but not active, set the return byte to **Media has not been changed**
- If the media byte in the new BPB does not match the old media byte, set the return byte to **Media has been changed**
- If the current volume ID matches the previous volume ID, set the return byte to **Don't know if media has been changed.**

DOS 3.00 to 3.30: If the driver has set the removable media bit 11 of the device header attribute word to 1 and the driver returns -1 (media changed), the driver must set the DWORD pointer to the previous volume identification field. If DOS determines that the media changed is an error, DOS generates an error 0FH (Invalid Disk Change) on behalf of the device. If the driver does not implement volume identification support, but has bit 11 set to 1, the driver should set a pointer to the string "NO NAME ", 0.

Media Descriptor Byte

Currently the media descriptor byte has been defined for a few media types. This byte should be identical to the media byte if the device has the non−IBM format bit off. These predefined values are:

Media descriptor byte —>	1	1	1	1	1	x	x	x
bits —>	7	6	5	4	3	2	1	0

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	

Note: An exception to the above bit meanings is that the media descriptor byte value of F0 is used to indicate any media types not defined.

Examples of current DOS media descriptor bytes:

Disk Type	# Sides	sectors/ track	Media Descriptor
Fixed disk	--	--	F8H
5 1/4 inch	2	15	F9H
5 1/4 inch	1	9	FCH
5 1/4 inch	2	9	FDH
5 1/4 inch	1	8	FEH
5 1/4 inch	2	8	FFH
8 inch	1	26	FEH
8 inch	2	26	FDH
8 inch	2	8	FEH
3 1/2 inch	2	9	F9H
3 1/2 inch	2	18	F0H

Notes:

The two media descriptor bytes that are the same for 8 inch diskettes (FEH) is not a misprint. To determine whether you are using a single sided or a double-sided diskette, attempt to read the second side, and if an error occurs you can assume the diskette is single sided.

1. Media descriptor F0H may be used for those media types not described above.

2. These media descriptor values are provided as a reference. Programs should not use these values.

3. DOS internal routines use information in the BIOS parameter block (BPB) to determine the media type of IBM formatted diskettes rather than using these values. These media descriptor bytes can no longer be guaranteed to indicate a unique media type.

For 8−inch diskettes:

FEH (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.

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

FEH Double 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.

BUILD BPB (BIOS Parameter Block)

Command code = 2

ES:BX

Field	Length
Request header	13 – BYTE
Media descriptor from DOS	BYTE
Transfer address (buffer address)	DWORD
Pointer to BPB table	DWORD

DOS calls BUILD BPB under the following two conditions:

- If "Media Changed" is returned.
- If "Not Sure" is returned, there are no used buffers. Used buffers are buffers with changed data not yet written to the disk.

The driver must perform the following:

- Set the pointer to the BPB.
- Set the status word in the request header.

The device driver must determine the media type that is currently in the unit to return the pointer to the BPB table. In previous versions of IBMBIO, the FAT ID byte determined the structure and layout of the media. Since the FAT ID byte has only eight possible values (F8 through FF), it is clear that, as new media types are invented, the available values will soon be exhausted. With the varying media layouts, DOS needs to be aware of the location of the FATs and directories before it requests to read them.

The following paragraphs explain the new method DOS will use to determine the media type.

The attribute word for the device driver will no longer indicate that FAT IDs are used to determine the media type, unless the media is formatted by a DOS 1.00 or 1.10 system. In this case the Attribute word indicates that the FAT ID is used.

The device driver checks the boot sector. If the boot sector is for DOS 2.00 to 3.30, the BPB from the boot sector is returned. If the boot sector is not for DOS 2.00 to 3.30, the device driver reads the first sector of the FAT to get the FAT ID. The FAT ID is examined and the corresponding BPB is returned.

In the latter case the media was formatted by a DOS 1.00 or a 1.10 system, so the FAT ID byte is used to determine the media type. Only two formats are possible for diskettes formatted by a 1.00 or 1.10 system: 5 1/4-inch single-sided (FEH) and 5 1/4-inch double sided (FFH.)

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:

For DOS 2.10, 3 BYTE near JUMP (E9H) or for DOS 3.00 to 3.30, 2 BYTE short JUMP (EBH) followed by a NOP (90H)
8 BYTES 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 intended to help the device driver understand the media. The number of heads is useful for supporting different multihead 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.

For drivers that support volume identification and disk change, this call should cause a new volume identification to be read off the disk. This call indicates that the disk has legally changed.

INPUT or OUTPUT

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

ES:BX

Field	Length
Request header	13 – BYTE
Media descriptor byte	BYTE
Transfer address (buffer address)	DWORD
Byte/sector count	WORD
Starting sector number (no meaning on character devices)	WORD
For DOS 3.00 to 3.30, pointer to the volume identification if error code 0FH is returned	DWORD

The driver must perform the following:

- Set the status word in the request header.
- Perform the requested function.
- Set the actual number of sectors (or bytes) transferred.

  Note: No error checking is performed on an IOCTL I/O call. However, the driver must set the return sector (byte) count to the actual number of bytes 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, ignores 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.

If the driver returns an error code of 0FH (Invalid Disk Change), it must put a DWORD pointer to an ASCIIZ string which is the correct volume identification to ask the user to reinsert the disk.

DOS 3.00 to 3.30: The reference count of open files on the disk (maintained by OPEN and CLOSE calls) allows the driver to determine when to return error 0FH. If there are no open files (reference count = 0) and the disk has been changed, the I/O is all right, and error 0FH is not returned. If there are open files (reference count > 0) and the disk has been changed, an error 0FH situation may exist.

NONDESTRUCTIVE INPUT NO WAIT

Command code = 5

ES:BX

Field	Length
Request header	13 – BYTE
Byte read from device	BYTE

The driver must perform the following:

- Return a byte from the device.
- Set the status word in the request header.

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 *nondestructive input*). This call allows DOS to look ahead one input character.

STATUS

Command codes = 6 and 10

ES:BX

Field	Length
Request header	13 – BYTE

The driver must perform the following:

- Perform the requested function.
- Set the busy bit.
- Set the status word in the request header.

The busy bit is set as follows:

For output on character devices— if the busy bit is 1 on return, a write request would wait for completion of a current request. If the busy bit is 0, there is no current request. Therefore, a write request would start immediately.

For input on character devices with a buffer— if the busy bit is 1 on return, a read request goes to the physical device. If the busy bit is 0, there are characters in the device buffer and a read returns quickly. It also indicates that the user has typed something. DOS assumes that all character devices have a type – ahead input buffer. Devices that do not have this buffer should always return busy = 0 so that DOS does not hang waiting for information to be put in a buffer that does not exist.

FLUSH

Command codes = 7 and 11

ES:BX

Field	Length
Request header	13 – BYTE

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.

OPEN or CLOSE (DOS 3.00 to 3.30)

Command codes = 13 and 14

ES:BX

Field	Length
Static request header	13 – BYTE

These calls are designed to give the device information about current file activity on the device if bit 11 of the attribute word is set. On block devices, these calls can be used to manage local buffering. The device can keep a reference count. Every OPEN causes the device to increment the reference count. Every CLOSE causes the device to decrement the reference count. When the reference count is 0, it means there are no open files on the device. Therefore, the device should flush buffers inside the device that it has written to because now the user can change the media on a removable media drive. If the media has been changed, it is advisable to reset the reference count to 0 without flushing the buffers. This can be thought of as "last close causes flush."

These calls are more useful on character devices. The OPEN call can be used to send a device an initialization string. On a printer, this could cause a string to be sent that would set the the font, the page size, etc., so that the printer would always be in a known state at the start of an I/O stream. Similarly the CLOSE call can be used to send a post string (like a form feed) at the end of an I/O stream. Using IOCTL to set these pre and post strings provides a flexible mechanism of serial I/O device stream control.

Note: Since all processes have access to STDIN, STDOUT, STDERR, STDAUX, and STDPRN (handles 0,1,2,3,4), the CON, AUX, and PRN devices are always open. IBM DOS calls this function in response to interrupt 21H calls 0FH, 10H, 3DH and 3EH only if file sharing (SHARE.EXE) is loaded.

REMOVABLE MEDIA (DOS 3.00 to 3.30)

Command code = 15

ES:BX

Field	Length
Static request header	13 – BYTE

To use this call, set bit 11 of the attribute field to 1. Block devices can only use this call through a subfunction of the IOCTL function call (44H). This call is useful because it allows a utility to know whether it is dealing with a nonremovable media drive or with a removable media drive. For example, the FORMAT utility needs to know whether a drive is removable or nonremovable because it prints different versions of some prompts.

The information is returned in the BUSY bit of the status word. If the busy bit is 1, the media is nonremovable. If the busy bit is 0, the media is removable.

Note: No error bit checking is performed. It is assumed that this call always succeeds.

Generic IOCTL Request (DOS 3.20 and 3.30)

Command code = 19

ES:BX

Field	Length
Static request header	13 – BYTE
Major function	BYTE
Minor function	BYTE
Contents of SI	WORD
Contents of DI	WORD
Pointer to Generic IOCTL request packet	DWORD

The driver must:

- Support the functions described under Generic IOCTL request
- Maintain its own track table (TrackLayout.)

DOS 3.20 and 3.30 use call AL = 0DH (Generic IOCTL request) to get or set device parameters and to format and verify a track on a logical device. DOS 3.30 uses call AL = 0CH (Generic IOCTL request) to implement codepage switching. The functions supported by Generic IOCTL request are explained in Chapter 6 under IOCTL 44H.

Get Logical Device (DOS 3.20 and 3.30)

Command code = 23

ES:BX

Field	Length
Static request header	13 – BYTE
Input (unit code)	BYTE
Command code	BYTE
Status	WORD
Reserved	DWORD

Set Logical Device (DOS 3.20 and 3.30)

Command code = 24

ES:BX

Field	Length
Static request header	13 – BYTE
Input (unit code)	BYTE
Command code	BYTE
Status	WORD
Reserved	DWORD

The CLOCK$ Device

A popular feature is a "Real Time Clock" board. To allow this board to be integrated into the system for TIME and DATE, there is a special device (determined by the attribute word) which is the CLOCK$ device. This device defines and performs functions like any other character device (most functions will be set done bit, reset error bit, return). When a read or write to this device occurs, exactly 6 bytes are transferred. The first 2 bytes are a word, which is the count of days since 1 – 1 – 80. The third byte is minutes; the fourth is hours; the fifth 1/100 is seconds; and the sixth is seconds. Reading the CLOCK$ device gets the date and time, writing to it sets the date and time.

Sample Device Driver

The Supplemental diskettes for DOS versions 3.00 to 3.20 contain a sample device driver listing called VDISK.LST. For DOS version 3.30, the sample device driver listing is called VDISK.ASM and it is on the Technical Reference Utilities diskette. Use the PRINT command to print a copy of the listing for reference.

Chapter 3. Using Extended Screen and Keyboard Control

Introduction

This chapter explains how you can issue special control character sequences to:

- Control the position of the cursor
- Erase text from the screen
- Set the mode of operation
- Redefine the meaning of keyboard keys

Control Sequences

The control sequences are valid if you issue them through DOS function calls that use standard input, standard output, or standard error output devices. These are the function calls 01H, 02H, 06H, 07H, 09H, 0AH, 15H, 22H, 28H and 40H.

The extended screen and keyboard control device driver ANSI.SYS must be installed by placing the following statement in the configuration file CONFIG.SYS:

device = [*d*:][*path*]ansi.sys

The size of DOS in memory increases by the size of ANSI.SYS.

Control Sequence Syntax

Each of the cursor control sequences is in the format:

ESC [*parameters* COMMAND

ESC	The 1 – byte ASCII code for ESC (1BH). It is not the three characters ESC.
[	The character [.
parameters	The numeric values you specify for #. The # represents a numeric parameter. A numeric parameter is an integer value specified with ASCII characters. If you do not specify a parameter value, or if you specify a value of 0, the default value for the parameter is used.
COMMAND	An alphabetic string that represents the command. It is case specific.

For example:

```
ESC [2;10H
```

could be created using BASIC as follows:

```
The IBM Personal Computer Basic
Version 3.00 Copyright IBM Corp. 1981, 1984
xxxxx Bytes free

Ok
open "sample" for output as 1
Ok
print #1, CHR$(27);"[2;10H";"x row 2 col 10"
Ok
close #1
Ok
```

Notice that "CHR$(27)" is ESC.

Cursor Control Sequences

The following tables contain the cursor control sequences you can use to control cursor positioning.

Cursor Position

Cursor Position	Function
ESC [#;#H	Moves the cursor to the position specified by the parameters. The first parameter specifies the row number and the second parameter specifies the column number. The default value is 1. If no parameter is given, the cursor is moved to the home position.

This example copies the file SAMPLE from the previous example, to CON, which places the cursor on row 2 column 10 of the screen:

```
type sample
```

Cursor Up

Cursor Up	Function
ESC [#A	Moves the cursor up one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. This sequence is ignored if the cursor is already on the top line.

Cursor Down

Cursor Down	Function
ESC [#B	Moves the cursor down one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. The sequence is ignored if the cursor is already on the bottom line.

Cursor Forward

Cursor Forward	Function
ESC [#C	Moves the cursor forward one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored if the cursor is already in the rightmost column.

Cursor Backward

Cursor Backward	Function
ESC [#D	Moves the cursor back one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored if the cursor is already in the leftmost column.

Horizontal and Vertical Position

Horizontal and Vertical Position	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 1. If no parameter is given, the cursor is moved to the home position.

Cursor Position Report

Cursor Position Report	Function
ESC [#;#R	The cursor sequence report 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.

Device Status Report

Device Status Report	Function
ESC [6n	The console driver outputs a cursor position report sequence on receipt of device status report.

Note: Do not use the Device Status Report as part of a prompt.

The following Pascal program tells ANSI.SYS to put the current cursor position (row and column) in the keyboard buffer. Then the program reads the cursor position from the keyboard buffer and displays the cursor position on the screen.

```
PROGRAM dsr(INPUT,OUTPUT);

  VAR
    f:FILE OF CHAR;
    key:CHAR;

  FUNCTION inkey:CHAR;              { read character  }
    VAR                             {  from the       }
      ch:CHAR;                      { keyboard buffer }
    BEGIN
      READ(f,ch);
      inkey:=ch
    END;

  BEGIN
    ASSIGN(f,'user');
    RESET(f);
        WRITE(CHR(27),'[6n');    { issue a DSR    }
        key:=inkey;              { read up to     }
        key:=inkey;              { first digit    }
        key:=inkey;              { of the row     }
        WRITE('row ',inkey,inkey,'  column ');
        key:=inkey;              { skip to column}
        WRITE(inkey,inkey)       { write column   }
  END.
```

Save Cursor Position

Save Cursor Position	Function
ESC [s	The current cursor position is saved. This cursor position can be restored with the restore cursor position sequence (see below).

Restore Cursor Position

Restore Cursor Position	Function
ESC [u	Restores the cursor to the value it had when the console driver received the save cursor position sequence.

Erasing

The following tables contain the control sequences you can use to erase text from the screen.

Erase in Display

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

Erase in Line

Erase in Line	Function
ESC [K	Erases from the cursor to the end of the line and includes the cursor position.

Mode of Operation

The following tables contain the control sequences you can use to set the mode of operation.

They are:

- Set Graphics Rendition (SGR)
- Set Mode (SM)

- Reset Mode (RM)

Set Graphics Rendition (SGR)	Function
ESC [#;...;#m	Sets the character attribute specified by the parameters. All following characters have the attribute according to the parameters 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 Canceled 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) **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) **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 resets wrap at end-of-line mode (characters past end-of-line are thrown away).

Keyboard Key Reassignment

The following table contains the control sequences you can use to redefine the meaning of keyboard keys.

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 0 (NULL) the first and second code make up an extended ASCII redefinition (see Chapter 6 for a list of extended ASCII codes).

Here are some examples:

To execute these examples, you can either:

- Create a file that contains the following statements and then use the TYPE command to display the file that contains the statement.
- Execute the command at the DOS prompt.

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

   Creating a File:

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

   At the DOS Prompt:

```
prompt $e[65;81p      A becomes Q
prompt $e[97;113p     a becomes q
prompt $e[81;65p      Q becomes A
prompt $e[113;97p     q becomes a
```

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

   Creating a File:

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

   At the DOS Prompt:

```
prompt $e[0;68;"dir";13p
```

   The $e is the prompt command characters for ESC. The 0;68 is the extended ASCII code for the F10 key; 13 decimal is a carriage return.

3. The following example sets the prompt to display the current directory on the top of the screen and the current drive on the current line.

```
prompt $e[s$e[1;30f$e[K$p$e[u$n$g
```

If the current directory is C:\FILES, and the current drive is C, this example would display:

```
                    C:\FILES

C>
```

4. The following assembly language program reassigns the F10 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     100H
ENTPT   JMP    SHORT START
STRING  DB     27,'[0;68;DIR B:";13p'
                              ;REDFINE F10 KEY
STRSIZ  EQU    $-STRING       ;LENGTH OF ABOVE
                              ;MESSAGE
HANDLE  EQU    1              ;PRE-DEFINED FILE

START   PROC   NEAR
        MOV    BX,HANDLE      ;STANDARD OUTPUT
                              ;DEVICE
        MOV    CX,STRSIZ      ;GET SIZE OF ABOVE
                              ;MESSAGE
        MOV    DX,OFFSET
               STRING         ;PASS OFFSET STRING
                              ;TO BE SENT
        MOV    AH,40H         ;FUNCTION="WRITE
                              ;TO DEVICE"
        INT    21H            ;CALL DOS
        RET                   ;RETURN TO DOS
START   ENDP

CSEG    ENDS
        END    ENTPT
```

Chapter 4. File Management Notes

Introduction

This chapter tells you how to:

- Use file management functions (FCB function calls and Handle function calls)
- Do file I/O in ASCII mode and Binary mode

Version Specific Information

The following information in this chapter is specific to a version of DOS:

Restrictions on FCB usage: For DOS 3.00 to 3.30, the number of files opened using FCBs is limited, if SHARE is loaded. The limit is set by the FCBS command in the CONFIG.SYS file.

File Management Functions

Use DOS function calls to create, open, close, read, write, rename, find, and erase files. There are two sets of function calls that DOS provides for support of file management. They are:

- File Control Block function calls (FCB function calls 0FH – 24H)
- Extended function calls (Handle function calls 39H – 62H)

Handle function calls are easier to use and more powerful than FCB function calls. The following table compares the use of FCB function calls to Handle function calls.

FCB Calls	**Handle Calls**
Addresses files that are only in the current directory.	Addresses files in *any* directory.
Requires that the application program maintain a file control block to open, create, rename, or delete a file. For I/O requests, the application program also needs an FCB.	Does not require maintenance of an FCB. Requires a string that contains the drive, path, and filename to open, create, rename, or delete a file. For file I/O requests, the application program only has to maintain a 16-bit word (file handle) that is supplied by DOS.

The only reason an application should use FCB function calls is to maintain the ability to run under DOS version 1.10. To do this, a program can only use function calls supplied by DOS 1.10 (00H - 2EH).

FCB Function Calls

FCB function calls require the use of one file control block per open file, which is maintained by the application program and DOS. The application program supplies a pointer to the FCB and fills in the appropriate fields required by the specific function call. An FCB function call can perform file management on any valid drive on the system, but only in the current directory of the specified drive. By using the current block, current record, and record length fields of the FCB, you can perform sequential I/O by using the sequential read or write function calls. Random I/O can be performed by filling in the random record and record length fields. See "File Control Block" on page 7-12 for information on the FCB structure.

Several possible uses of FCB type calls are considered programming errors and should not be done under any circumstances. This is to avoid problems with file sharing and compatibility. One such error occurs when a program uses the same FCB structure to access more than one open file. By opening a file using an FCB, doing I/O, and then replacing the filename field in the file control block with a new filename, a program can then open a second file using the same FCB. This is invalid because DOS writes control information about the file into the reserved fields of the FCB. This information is changed when the second file is opened using the same FCB. If the program then replaces the filename field with the original filename and then tries to perform I/O to this file, DOS may become confused because the control information has been changed. An FCB should never be used to open a second file without closing the file that is currently open. If more than one file is to be open concurrently, separate FCBs should be used.

A program should also never tamper with the DOS reserved fields in the FCB, as the contents and structure of these fields change in different versions of DOS. It is also good programming practice to close all files after all I/O to a file is done. This avoids potential file sharing problems that require a limit on the number of files concurrently open using FCB function calls. A delete or a rename on a file that is currently open is also considered an error and should not be attempted by an application program.

A program should not close a FCB file and continue writing to the file. This behavior is not supported if SHARE.EXE is loaded or the file is on a network. This behavior is never recommended.

Handle Function Calls

The recommended method of file management is by using the extended "handle" set of function calls. These calls are not restricted to files in the current directory. Also, the handle set of file management calls allow the application program to define the type of access that other processes can have concurrently with the same file if file sharing is loaded.

To create or open a file, the application supplies a pointer to an ASCIIZ string giving the name and location of the file. An ASCIIZ string contains an optional drive letter, optional path and mandatory file specification, terminated by a byte of 00H. The following is an example of an ASCIIZ string:

```
DB   "a:\path\filename.ext",0
```

If the file is being created, the application program also supplies the attribute of the file. This is a set of values that defines if the file is read only, hidden, system, directory, or volume label. See "DOS Disk Directory" on page 5-10 for information on file attributes.

If the file is being opened, the program can define the sharing and access modes that the file is opened in. The access mode informs DOS what operations your program will perform on this file (read – only, write – only or read/write). The sharing mode controls the type of operations other processes may perform concurrently on the file. A program can also control if a child process inherits the open files of the parent. The sharing mode field has meaning only if file sharing is loaded when the file is opened.

To rename or delete a file, the application program simply needs to provide a pointer to the ASCIIZ string containing the name and location of the file and another string with the new name if the file is being renamed.

The open or create function calls return a 16 – bit value referred to as the file handle. To do any I/O to a file, the program uses this handle to reference the file. Once a file is opened, a program no longer needs to maintain the ASCIIZ string pointing to the file, nor is there any requirement to stay in the same directory. DOS keeps track of the location of the file regardless of what directory is current.

Sequential I/O can be performed using the handle read (3FH) or write (40H) function calls. The offset in the file that I/O is performed to is automatically moved to the end of what was just read or written. If random I/O is desired, the LSEEK (42H) function call can be used to set the offset into the file that the I/O is performed at.

Special File Handles

DOS sets up five special file handles for use by application programs. These handles are:

0000H Standard input device (Stdin)

0001H Standard output device (Stdout)

0002H Standard error device (Stderr)

0003H Standard auxiliary device (Stdaux)

0004H Standard printer device (Stdprn)

These handles are predefined by DOS and can be used by any application program. They do not need to be opened by the program, although a program can close these handles. Stdin should be treated as a read – only file, and Stdout and Stderr should be treated as write only handles. Stdin and Stdout can be redirected. All handles inherited by a process can be redirected, but not at the command line.

These handles are very useful for doing I/O to and from the console device. For example, you could read input from the keyboard using the read (3FH) function call and file handle 0000H (Stdin), and write output to the console screen with the write function call (40H) and file handle 0001H (Stdout). If you wanted an output that could not be redirected, you could output it using file handle 0002H (Stderr). This is very useful for error messages or prompts that a user must see in order to act upon them.

File handles 0003H (Stdaux) and 0004H (Stdprn) can both be read from and written to. Stdaux is typically a serial device and stdprn is usually a parallel device.

ASCII and Binary Mode

I/O to files is done in binary mode. This means that the data is read to or written from a file without modification. However, DOS can also read or write to devices in ASCII mode. In ASCII mode, DOS does some string processing and modification to the characters read or written. The predefined handles are in ASCII mode when initialized by DOS. All other file handles that don't refer to devices are in binary mode. A program can use the IOCTL (44H) function call to set the mode that I/O is done to a device. The predefined file handles for are all devices, so the mode can be changed from ASCII to binary via IOCTL. Regular file handles that are not devices are always in binary mode, and they cannot be changed to ASCII mode.

The predefined handles Stdin (0000H), Stdout (0001H), and Stderr (0002H) are all duplicate handles. If the IOCTL function call is used to change the mode of any of these three handles, the mode of all three handles is changed. For example, if IOCTL was used to change Stdout to binary mode, then Stdin and Stderr would also be changed to binary mode.

File I/O in Binary Mode

When a file is read in binary mode:

- The characters ∧S (Scroll lock), ∧P (Print Screen), ∧C (Control Break) are not checked for during the read. Therefore, no printer echo occurs if ∧S or ∧P are read.
- There is no echo to Stdout (0001H).
- Reads the number of specified bytes and returns immediately when the last byte is received or the end of file is reached.
- Allows no editing of the line input using the function keys if the input is from Stdin (0000H).

When a file is written in binary mode:

- The characters ∧S,∧P,∧C are not checked for during the write operation. Therefore there is no printer echo.
- There is no echo to Stdout (0001H).
- The exact number of bytes specified are written.
- Does not caret control characters. For example, control D is sent out as byte 04H instead of the two bytes ∧ and D.
- Does not expand tabs into spaces.

File I/O in ASCII Mode

When a file is read in ASCII mode:

- Checks for the characters ∧C,∧S, and ∧P.

- Returns as many characters as there are in the device input buffer, or the number of characters requested, whichever is less. If the number of characters requested was less than the number of characters in the device input buffer, then the next read will address the remaining characters in the buffer.

- If there are no more bytes remaining in the device input buffer, read a line (terminated with ∧M) into the buffer. This line may be edited with the function keys. The characters returned terminate with a sequence of 0DH,0AH (∧M,∧J) if the number of characters requested is sufficient to include them. For example, if 5 characters were requested, and only 3 were entered before the carriage return (0DH or ∧M) was presented to DOS from the console device, the 3 characters entered and 0DH, and 0AH would be returned. However, if 5 characters were requested and 7 were entered before the carriage return, only the first 5 characters would be returned. No 0DH,0AH sequence would be returned in this case. If less than the number of characters requested are entered when the carriage return is received, the characters received and the 0DH,0AH would be returned. The reason the 0AH (line feed or ∧J) byte is added to the returned characters is to make devices look like text files.

- If a 1AH (∧Z) is found, the input is terminated at that point. No 0DH,0AH sequence is added to the string.

- Echoing is performed.

- Tabs are expanded into spaces on echo. They are left as a tab byte (09H) in the input buffer.

When a file is written in ASCII mode:

- The characters ∧S,∧P, and ∧C are checked for during the write operation.

- Expands tabs to 8−character boundaries and fills with spaces (20H).

- Carets control characters. For example, ∧D is written as two bytes, ∧ and D.

- Bytes are output until the the number specified is output or until a ∧Z is found. The number actually output is returned to the user.

Number of Open Files Allowed

The number of files that can be open concurrently is restricted by DOS. This number is determined by how the file is opened or created (FCB or handle function call) and the number specified by the FCBS and FILES commands in the CONFIG.SYS file. The number of files allowed open by FCB function calls and the number of files that can be opened by handle type calls are independent of one another.

Restrictions on FCB Usage

If file sharing is not loaded using the SHARE command, there are no restrictions on the number of files concurrently open using FCB function calls. However, when file sharing is loaded, the maximum number of FCB opened files is limited by the value set by the FCBS command in the CONFIG.SYS configuration file. For information on the FCBS command, refer to Chapter 4 of the *DOS Reference* for versions 3.00 to 3.30. The FCBS command has two values that you can specify *m, n*. The value for *m* specifies the total number of files that can be opened by FCBs, and the value for *n* specifies the

number of files opened by FCBS that are protected from being closed.

When the maximum number of FCB opens is exceeded, DOS automatically closes the least recently used file. Any attempt to access this file results in an interrupt 24H critical error message, "FCB not available." If this occurs while an application program is running, the value specified for *m* in the FCBS command should be increased.

When DOS determines the least recently used file to close, it does not include the first *n* files opened, therefore the first *n* are protected from being closed.

Restrictions on Handle Usage

The number of file handles that can be open at one time by all processes is determined by the FILES command in the CONFIG.SYS file (for more information see the FILES command in the *DOS Reference*).

The default number of handles available to a single process is twenty. In DOS 3.30, the maximum number of handles can be increased up to 64K by using Set Handle Count (function 67H).

Each open handle is associated with a single file or device. Several handles can reference the same file or device. This is why the maximum handle limit exceeds the maximum value of the FILES command in CONFIG.SYS. Each Open (function 3DH) to a file or device uses both a handle and a FILES= entry.

The effective limit to the number of Opens a program can issue is the the remaining handle count for that program or the remaining FILES= entries for the system, whichever is smaller. An Open can fail because it runs out of either handles or FILES= entries.

As stated earlier, five handles are automaticaly setup by DOS. This reduces the number of handles available to the program. If a program is using additional handles when it calls another program, the handles are passed to the new program.

The rules for relating handles to open files are listed below:

1. Each Open uses both a handle and one of the CONFIG.SYS FILES entries.

2. Each Duplicate (function 45H) uses a handle but not a CONFIG.SYS FILES item. A Duplicate increases the use count of the FILES= entry for that handle.

3. Each Exec (function 4BH) duplicates open handles which have the Inheritance flag set to zero.

4. Each Close (function 3EH) releases a handle and decrements the use count for the FILES= entry. If the count reaches zero, the FILES= entry is made available to the system.

5. A Terminate a Process (function 4CH) closes all handles of a program.

The sequence of events in the example below illustrates the relationship between handles and FILES= entries. Assume a FILES=10 statement in the CONFIG.SYS file.

Action taken by program or current state	Programs loaded	Handles used	FILES used
At DOS prompt just after boot	COMMAND	5	3
Start MYEDIT editor program	COMMAND	5	
	MYEDIT	5	3
Edit file XXX (kept open)	COMMAND	5	
	MYEDIT	6	4
Go to DOS command line	COMMAND	5	
	MYEDIT	6	
	COMMAND	6	4
TYPE file ZZZ	COMMAND	5	
	MYEDIT	6	
	COMMAND	7	5
Exit DOS	COMMAND	5	
	MYEDIT	6	4
File and exit MYEDIT	COMMAND	5	3
Run MYDBASE program, open 10 files	COMMAND	5	
Note that the 8th open fails	MYDBASE	13	10
Exit MYDBASE, closing all files	COMMAND	5	3

Allocating Space to a File

Files are not necessarily written sequentially on a disk. Space is allocated as it is needed and the next 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 may not be written in sequential sectors. However, due to the mapping (chaining) of file space via the File Allocation Table (FAT), and the function calls available, any file can be used in either a sequential or random manner.

Space is allocated in increments called clusters. Cluster size varies from a low of one sector of disk space per cluster on a single – sided diskette to a higher number of sectors/cluster on other disk formats. The cluster size of a fixed disk is based on the size of the DOS partition, and is determined when the fixed disk is formatted with the FORMAT command. For example, for a 10M byte fixed disk that is totally dedicated to one DOS partition, the cluster size is equal to 8 sectors.

An application program should not concern itself with the way that DOS allocates disk space to a file. The size of a cluster is only important in that it determines the smallest amount of space allocated to a file at one time. For example, a diskette with 2 sectors per cluster and a sector size of 512 bytes would allocate diskette space to a file in 1024 byte blocks. Therefore, even if a file was less than one cluster long, a cluster's worth of disk space would be allocated to the file. If more disk space is needed, additional clusters are allocated to the file. A disk is considered full when all the available clusters have been allocated to files.

Chapter 5. DOS Disk Allocation

Introduction

This chapter contains the following information about DOS:

- The boot record
- The DOS file allocation table (FAT) for 12-bit and 16-bit FATs
- The DOS disk directory
- The data area

Version Specific Information

The following information in this chapter is specific to a version of DOS:

DOS File Allocation Table (FAT):

- 12-bit FATs are for use with DOS versions 2.10 and 3.00 to 3.30.
- 16-bit FATs are for use with DOS versions 3.00 to 3.30.

Also, for DOS versions 3.00 to 3.30, the File Allocation Table indicator F9H is used to identify 15 sector-per-track diskettes.

The DOS Area

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 - 1 sector
First copy of file allocation table (FAT) - variable size
Second copy of file allocation table - same size as first copy of FAT
Root directory - variable size
Data area

The following sections describe each of the allocated areas.

The Boot Record

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

File Allocation Table (FAT)

This section explains how DOS uses the file allocation table (FAT) to convert the clusters of a file to logical sector numbers. We recommend that system utilities use the DOS handle function calls rather than interpreting the FAT.

The 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 or a 16-bit entry (2 bytes) when a fixed disk has more than 20740 sectors as is the case for fixed disks larger than 10M bytes.

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 FAT can be in a 12-bit or a 16-bit format. DOS determines whether a disk has a 12- or 16-bit FAT by looking at the total number of allocation units on the disk. For all diskettes and fixed disks with DOS partitions less than 20740 sectors, the FAT uses a 12-bit value to map a cluster. For larger partitions, DOS uses a 16-bit value.

The number of sectors for a disk can be determined by using the formula:

TS = SPT * H * C.

Where:

TS is the total number of sectors on the disk.

SPT is the number of sectors-per-track or per-cylinder.

H is the number of heads.

C is the number of cylinders.

The number of sectors for a 10MB IBM fixed disk is 20740 (17 * 4 * 305).

The second, third, and fourth (if applicable for 16-bit FATs) bytes always contain FFFFH. 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.
F9	Dual sided, 15 sector-per-track diskette (1.2 MB).
F9	Dual sided, 9 sector-per-track diskette (720 KB).
F8	Fixed disk.
F0	Others.

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

These values are provided as a reference. Therefore, programs should not make use of these values.

Note: DOS internal routines use information in the BIOS parameter block (BPB) to determine the media type of IBM formatted diskettes rather than using these values. These media descriptor bytes can no longer be guaranteed to indicate a unique media type.

Each entry contains 3 hexadecimal characters, (or 4 for 16-bit FATs). ( ) indicates the high-order four bit value in the case of the 16-bit FAT entries. They can be either:

Hex Value	Meaning
(0)000	if the cluster is unused and available, or
(F)FF8--(F)FFF	to indicate the last cluster of a file, or
(X)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.

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

The file allocation table always occupies the sector or sectors immediately following the boot record. If the FAT is larger than 1 sector, the sectors occupy consecutive sector 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.).

How to Use the File Allocation Table for 12 – Bit FAT Entries

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 (FF8-FFF)H, no more clusters are 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 25H and 26H and by DEBUG):

1. Subtract 2 from the cluster number.

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

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

How to Use the File Allocation Table for 16−Bit FAT Entries

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 used by 2 (each FAT entry is 2 bytes long).

2. Use MOV word instruction to move the word at the calculated FAT offset into a register.

3. If the resultant 16 bits are (FFF8-FFFF)H, no more clusters are in the file. Otherwise, the 16 bits contain the cluster number of the next cluster in the file.

DOS Disk Directory

The FORMAT command 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.

Directory Entries

Since directories other than the root directory are actually files, there is no limit to the number of entries they may contain.

All directory entries are 32 bytes long, and are in the following format (byte offsets are in decimal). The following paragraphs describe the directory entry bytes:

Bytes 0-7

Bytes 0 through 7 represent the filename. The first byte of the filename indicates the status of the filename. The status of a filename can contain the following values:

00H Filename never used. This is used to limit the length of directory searches, for performance reasons.

05H Indicates that the first character of the filename actually has an E5H character.

E5H Filename was used, but the file has been erased.

2EH The entry is for a directory. If the second byte is also 2EH, the cluster field contains the cluster number of this directory's parent directory (0000H if the parent directory is the root directory).

Any other character is the first character of a filename.

Bytes 8-10

These bytes indicate the filename extension.

Byte 11

This byte indicates the file's attribute. The attribute byte is mapped as follows (values are in hexadecimal):

Note: The volume label and subdirectory bits of an attribute cannot be changed using CHMOD.

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.

01H Indicates that the file is marked read-only. An attempt to open the file for output using function call 3DH results in an error code being returned. This value can be used with other values below.

02H Indicates a hidden file. The file is excluded from normal directory searches.

04H Indicates a system file. The file is excluded from normal directory searches.

08H Indicates that 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.

10H Indicates that the entry defines a subdirectory and is excluded from normal directory searches.

20H Indicates an 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 the file was changed since it was last backed up. This bit can be used along with other attribute bits.

All other bits are reserved, and must be 0.

Bytes 12-21

This is a reserved area by DOS.

Bytes 22-23

These bytes contain the time when the file was created or last updated. The time is mapped in the bits as follows:

```
<          23          >  <          22          >
15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
h  h  h  h  h  m  m  m  m  m  m  x  x  x  x  x
```

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.

Bytes 24-25

This area contains the date when 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.

Bytes 26-27

This area contains the starting cluster number of the first cluster in the file. 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 "File Allocation Table (FAT)" for details about converting cluster numbers to logical sector numbers.

Bytes 28-31

This area contains the 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.

The Data Area

Allocation of space for a file (in the data area) is done only when needed (it is not preallocated). 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 multisided 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 (5 1/4)	8	1	4	64	1
2 (5 1/4)	8	1	7	112	2
1 (5 1/4)	9	2	4	64	1
2 (5 1/4)	9	2	7	112	2
2 (5 1/4)	15	7	14	224	1
2 (3 1/2)	9	3	7	112	2
2 (3 1/2)	18	9	14	224	1

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 is 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 back to the description of the DOS File Allocation Table in this chapter for more information.

Chapter 6. DOS Interrupts and Function Calls

Introduction

This chapter contains:

- A list of the registers used by DOS.
- A list of the extended ASCII codes.
- A detailed description of all the interrupts and function calls.

Version Specific Information

The following information in this chapter is specific to a version of DOS:

Interrupts:

DOS version 2.10 supports interrupts 20H to 27H.

DOS versions 3.00 to 3.30 support interrupts 20H to 2FH.

Function Calls:

DOS version 2.10 supports function calls 00H to 57H.

DOS version 3.00 to 3.30 supports function calls 00H to 5CH and 62H. This includes the following function calls for DOS 3.00.

- 3DH Open File; supports file sharing
- 4408H Check if Device is Removable
- 440BH Change Sharing Retry Count
- 59H Get Extended Error
- 5AH Create Temporary File
- 5BH Create New File
- 5CH Lock/Unlock File Access
- 62H Get Program Segment Prefix Address

DOS versions 3.10 to 3.30 support function calls 00H to 62H, which include the following new function calls for DOS 3.10.

- 4409H Check if Device is Local or Remote
- 440AH Check if Handle is Local or Remote
- 5E00H Get Machine Name
- 5E02H Set Printer Setup
- 5E03H Get Printer Setup
- 5F02H Get Redirection List Entry
- 5F03H Redirect Device
- 5F04H Cancel Redirection

The following new function calls are supported by DOS 3.20:

- 440DH Generic IOCTL
- 440EH Get Drive Assignment
- 440FH Set Next Logical Drive Letter

The following new function calls are supported by DOS 3.30:

- 440CH Code page Switching
- 65H Get Extended Country Information
- 66H Get/Set Global Code Page

- 67H Set Handle Count
- 68H Commit File

For DOS 3.00 to 3.30, when an interrupt 24H (Critical Error Handler Vector) occurs, bits 3 – 5 of AH indicate which error responses are valid. Also, these versions of DOS handle invalid responses differently than DOS 2.10. Refer to "Handling of Invalid Responses (DOS 3.00 to 3.30)" in this chapter for more information.

DOS Registers

DOS uses the following registers, pointers, and flags when it executes interrupts and function calls.

Register Definition	General Registers
AX AH AL	Accumulator (16 – bit) Accumulator high – order byte (8 – bit) Accumulator low – order byte (8 – bit)
BX BH BL	Base (16 – bit) Base high – order byte (8 – bit) Base low – order byte (8 – bit)
CX CH CL	Count (16 – bit) Count high – order byte (8 – bit) Count low – order byte (8 – bit)
DX DH DL	Data (16 – bit) Data high – order (8 – bit) Data low – order (8 – bit)
Flags	OF,DF,IF,TF,SF,ZF,AF,PF,CF

Register Definition	Pointers
SP	Stack pointer (16 – bit)
BP	Base pointer (16 – bit)
IP	Instruction pointer (16 – bit)

Register Definition	Segment Registers
CS	Code segment (16 – bit)
DS	Data segment (16 – bit)
SS	Stack segment (16 – bit)
ES	Extra segment (16 – bit)

Register Definition	Index Registers
DI	Destination index (16 – bit)
SI	Stack index (16 – bit)

Register numbering convention: Each register is 16 bits long and is divided into a high and low byte. Each byte is 8 bits long. The bits are numbered from right to left. The low byte contains bits 0 through 7 and the high byte contains bits 8 through 15. The chart below shows the hexadecimal values assigned to each bit.

	High Byte	Low Byte
Bit	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
Hex value	8 4 2 1 8 4 2 1	8 4 2 1 8 4 2 1

Extended ASCII Codes

For certain keys or key combinations that cannot be represented in standard ASCII code, an extended ASCII code is returned. The extended ASCII code is returned as the second byte of a 2 byte string. Therefore, if the ASCII value returned is zero, examine the second byte to obtain the extended ASCII code.

The following table lists the extended ASCII codes and their meanings.

Extended ASCII Code	Meaning
3	NUL (null character)
15	Shift tab
16-25	Alt- Q, W, E, R, T, Y, U, I, O, P
30-38	Alt - Z, X, C, V, B, M, N
59-68	Function keys F1 through F10
71	Home
72	Cursor up
73	Page up
75	Cursor left
77	Cursor right
79	End
80	Cursor down
81	Page down

Extended ASCII Code	Meaning
82	Insert
83	Delete
84-93	F11-F20 (Shift F1-F10)
94-103	F21-F30 (Ctrl F1-F10)

Interrupts

If you want a program to examine or set the contents of any interrupt vector, use the DOS function calls (35H and 25H) provided for those purposes, and avoid referencing the interrupt vector locations directly.

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

20H Program Terminate

Issue interrupt 20H to exit from a program. This vector transfers to the logic in DOS to restore the terminate address, the Ctrl – Break address, and the critical error exit address to the values they had on entry to the program. All file buffers are flushed and all handles are closed. You should close all files changed in length (see function call 10H and 3EH) before issuing this interrupt. If the changed file is not closed, its length, date, and time are not recorded correctly in the directory.

For a program to pass a completion code or an error code when terminating, it must use either function call 4CH (Terminate a Process) or 31H (Terminate Process and Stay Resident). These two methods are preferred over using interrupt 20H, and the codes returned by them can be interrogated in batch processing. See function call 4CH for information on the ERRORLEVEL subcommand of batch processing.

Important: Before you issue interrupt 20H, your program must ensure that the CS register contains the segment address of its program segment prefix.

21H Function Request

Refer to "Function Calls" on page 6-36

22H Terminate Address

Control transfers to the address at this interrupt location when the program terminates. This address is copied into the program's Program Segment Prefix at the time the segment is created. Do not issue this interrupt directly, the EXEC function call does this for you.

23H Ctrl – Break Exit Address

If the user enters Ctrl – Break during standard input, standard output, standard printer, or asynchronous communications adapter operations, an interrupt 23H is executed. If BREAK is on, the interrupt 23H is checked on *most* function calls (except calls 06H and 07H). If the user written Ctrl – Break routine saves all registers, it may end with a return – from – interrupt instruction (IRET) to continue program execution. If the user – written interrupt 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, the program is aborted, otherwise execution continues (as with a return by IRET). If the user – written Ctrl – Break interrupt uses functions calls 09H or 0AH, then ∧C, carriage – return and linefeed are output. If execution is 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. Do not issue this interrupt directly.

24H 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 next page).

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

Error Code	Error Name
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 is in effect and contains the following from top to bottom:

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

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

(AL) = 0 ignore the error.
= 1 retry the operation.
= 2 terminate the program through interrupt 23H.
= 3 fail the system call that is in progress.

Note: Be careful when choosing ignore as a response because this causes DOS to believe that an operation has completed successfully when actually it may not have.

To return control from the critical error handler to a user error routine, the following should be true:

Before an INT 24H occurs:

1. The user application initialization code should save the INT 24H vector and replace the vector with one pointing to the user error routine.

When the INT 24H occurs:

2. When the user error routine receives control, it should push the flag register onto the stack, and then execute a CALL FAR to the original INT 24H vector saved in step 1.

3. DOS gives the appropriate prompt, and waits for the user input (Abort, Retry, Fail or Ignore). After the user input, DOS returns control to the user error routine at the instruction following the CALL FAR.

4. The user error routine can now do any tasks necessary. To return to the original application at the point the error occurred, the error routine needs to execute an IRET instruction. Otherwise, the user error routine should remove the IP, CS, and Flag registers from the stack. Control can then be passed to the desired point.

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
0 1 file allocation table
1 0 directory
1 1 data area

AH bits 3 – 5 indicate which responses are valid. They are:

Bit 3 = 0 if FAIL is not allowed
= 1 if FAIL is allowed
Bit 4 = 0 if RETRY is not allowed
= 1 if RETRY is allowed
Bit 5 = 0 if IGNORE is not allowed
= 1 if IGNORE is allowed

Handling of Invalid Responses (DOS 3.00 to 3.30)

If IGNORE is specified (AL = 0) and IGNORE is not allowed (bit 5 = 0), make the response FAIL (AL = 3).

If RETRY is specified (AL = 1) and RETRY is not allowed (bit 4 = 0), make the response FAIL (AL = 3).

If FAIL is specified (AL = 3) and FAIL is not allowed (bit 3 = 0), make the response ABORT (AL = 2).

Other Errors

If AH bit 7 = 1, 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 is involved, the contents of AL are unpredictable, the error code is in DI as above.

Notes:

1. Before giving this routine control for disk errors, all DOS versions from 3.00 to 3.30 perform five retries, except when they are actually in the FAT or a directory entry, then, they perform three retries.

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

3. This routine is entered in a disabled state.

4. All registers must be preserved.

5. This interrupt handler should refrain from using DOS function calls. If necessary, it may use calls 01H through 0CH, 30H, and 59H. Use of any other call destroys the DOS stack and leaves DOS in an unpredictable state.

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

7. If the interrupt handler handles errors itself rather than returning to DOS, it should restore the application program's registers from the stack, remove all but the last three words on the stack, then issue an IRET. This will return to the program immediately after the INT 21H that experienced the error. Note that if this is done, DOS will be in an unstable state until a function call higher than 0CH is issued; therefore, it is not recommended.

   The use of fail and then testing the extended error code of the INT 21H is the recommended way to end a critical error.

8. For DOS 3.00 to 3.30, IGNORE requests (AL = 0) are converted to FAIL for critical errors that occur on FAT or DIR sectors.

9. Refer to "Error Return Information" on page 6-40 and "Extended Error Codes" on page 6-44 for information on how to obtain additional error information.

10. For DOS 3.10 to 3.30, IGNORE requests (AL = 0) are converted to FAIL requests for network critical errors (50-79).

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

DWORD Pointer to next device (FFFFH if last device)
WORD Attributes: Bit 15 = 1 if character device. = 0 if block device If bit 15 is 1: Bit 0 = 1 if current standard input Bit 1 = 1 if current standard output Bit 2 = 1 if current NULL 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, 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 8 bytes starting at BP:SI + 10.

25H Absolute Disk Read

This transfers control directly to the device driver. On 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 is 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. Note that the mapping is different from that used by DOS version 1.10 for dual-sided diskettes.

All registers except the segment registers are destroyed by this call. If the transfer was successful, the carry flag (CF) is zero. If the transfer was not successful CF = 1 and (AX) 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 interrupt 24H is issued, and (AH) contains:

80H	Attachment failed to respond
40H	SEEK operation failed
08H	Bad CRC on diskette read
04H	Requested sector not found
03H	Write attempt on write-protected diskette
02H	Error other than types listed above

26H Absolute Disk Write

This vector is the counterpart of interrupt 25H above. Except that this is a write, the description above applies.

27H Terminate but Stay Resident

This vector is used by programs that are to remain resident when COMMAND.COM regains control.

DOS function call 31H is the preferred method to cause a program to remain resident, because this allows return information to be passed, and allows a program larger than 64K to remain resident. After initializing itself, the program must set DX to its last address plus one relative to the program's initial DS or ES value (the offset at which other programs can be loaded), then execute an interrupt 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 that are loaded into the high end of memory.

2. This interrupt restores the interrupt 22H, 23H, and 24H vectors in the same manner as interrupt 20H. 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. Memory can be more efficiently used if the block containing a copy of the environment is deallocated before terminating. This can be done by loading ES with the segment contained in 2C of the PSP, and issuing function call 49H (Free Allocated Memory).

5. DOS function call 4CH allows the terminating program to pass a completion (or error) code to

DOS, which can be interpreted within batch processing (see function call 31H).

6. Terminate but stay resident programs do not close files.

28H – 2EH Reserved for DOS

These interrupts are reserved for DOS use.

2FH Multiplex Interrupt

Interrupt 2FH is the multiplex interrupt. A general interface is defined between two processes. It is up to the specific application using interrupt 2FH to define specific functions and parameters.

Every multiplex interrupt handler is assigned a specific multiplex number. The multiplex number is specified in the AH register. The specific function that the handler is to perform is specified in the AL register. Other parameters are placed in the other registers, as needed. The handlers are chained into the interrupt 2FH interrupt vector. There is no predefined method for assigning a multiplex number to a handler. You must just pick one. To avoid a conflict if two applications choose the same multiplex number, the multiplex numbers used by an application should be patchable.

The multiplex numbers AH = 0 through AH = 7FH are reserved for DOS. Applications should use multiplex numbers C0H through FFH.

> **Note:** When in the chain for interrupt 2FH, if your code calls DOS or if you execute with interrupts enabled, your code must be reentrant/recursive.

Function Codes

AH = 1

AH = 1 is the resident part of PRINT.

The following table contains the function codes that you can specify in AL to request the resident portion of print to perform a specific function.

Function Codes	Description
0	Get installed state
1	Submit file
2	Cancel file
3	Cance all files
4	Status
5	End of status

Print Error Codes

The following table contains the error codes that are returned from the resident portion of print.

Error Codes	Description
1	Invalid function
2	File not found
3	Path not found
4	Too many open files
5	Access denied
8	Queue full
9	Busy
12	Name too long

Error Codes	Description
15	Invalid drive

AL = 0 Get Installed State

This call must be defined by all interrupt 2FH handlers. It is used by the caller of the handler to determine if the handler is present. On entry, AL = 0, AH = 1. On return, AL contains the installed state as follows:

AL = 0 Not installed, O.K. to install

AL = 1 Not installed, not O.K. to install

AL = FF Installed

AL = 1 Submit File

On entry, AL = 1, AH = 1, and DS:DX points to the submit packet. A submit packet contains the level (BYTE) and a pointer to the ASCIIZ string (DWORD in offset segment form). The level value for DOS 3.10 to 3.30 is 0. The ASCIIZ string must contain the drive, path, and filename of the file you want to print. The filename cannot contain global filename characters.

AL = 2 Cancel File

On entry, AL = 2, AH = 1, and DS:DX points to the ASCIIZ string for the print file you want to cancel. Global filename characters are allowed in the filename.

AL = 3 Cancel all Files

On entry, AL = 3 and AH = 1.

AL = 4 Status

This call holds the jobs in the print queue so that you can scan the queue. Issuing any other code releases the jobs. On entry, AL = 4, AH = 1. On return, DX contains the error count. The error count is the number of consecutive failures PRINT had while trying to output the last character. If there are no failures, the number is zero. DS:SI points to the print queue. The print queue consists of a series of filename entries. Each entry is 64 bytes long. The first entry in the queue is the file currently being printed. The end of the queue is marked by a queue entry having a null as the first character.

AL = 5 End of Status

Issue this call to release the queue from call 4. On entry, AL = 5 and AH = 1. On return, AX contains the error codes. For information on the error codes returned, refer to "Print Error Codes" on page 6-29.

AL = F8 – FF Reserved by DOS

AH = 2

AH = 2 is the resident part of ASSIGN. The Get Installed State function (AL = 0) is supported.

AH = 10H

AH = 10H is the resident part of SHARE. The GET Installed State function (AL = 0) is supported.

AH = B7H

AH = B7 is the resident part of APPEND. The Get Installed State function is also supported.

Example 2FH Handler

```
MYNUM           DB      X ; X = The specific AH
                          ; multiplex number.
INT_2F_NEXT     DD      ?  ; Chain location
INT_2F:

ASSUME DS:NOTHING,ES:NOTHING,SS:NOTHING

  CMP       AH,MYNUM
  JE        MINE
  JMP       INT_2F_NEXT    ; Chain to next
                           ;2FH Handler

MINE:

  CMP       AL,0F8H
  JB        DO_FUNC
  IRET                     ; IRET on reserved
                           ; functions

DO_FUNC:

  OR        AL,AL
  JNE       NON_INSTALL    ; Non Get Installed
            ; State request
  MOV       AL,0FFH        ; Say I'm here
  IRET                     ; All done

NON_INSTALL:
.
.
.
```

Installing the Handler

```
    MOV     AH,MYNUM
    XOR     AL,AL
    INT     2FH                     ; Ask if already
                                    ; installed
    OR      AL,AL
    JZ      OK INSTALL

BAD INSTALL:                        ; Handler already
                                    ; installed


OK_INSTALL:                          ; Install my
                                     ; handler

    MOV     AL,2FH
    MOV     AH,GET_INTERRUPT VECTOR
    INT     21H                     ; Get multiplex
                                    ; vector
    MOV     WORD PTR INT_2F NEXT+2,ES
    MOV     WORD PTR INT_2F_NEXT,BX
    MOV     DX,OFFSET INT_2F
    MOV     AL,2FH
    MOV     AH,SET_INTERRUPT_VECTOR
    INT     21H                     ; Set multiplex
                                    ; vector
.
.
.
```

30H-3FH Reserved for DOS

These interrupts are reserved for DOS use.

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):

Hex Values	Meaning
0	Program terminate
1-C	Traditional character device I/O
D-24	Traditional file management
25-26	Traditional nondevice functions
27-29	Traditional file management
2A-2E	Traditional nondevice 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
59-5C	Extended function group
5E-5F	Network function group
62	Extended function group

Listing of Function Calls

00H Program terminate
01H Keyboard input
02H Display output
03H Auxiliary input
04H Auxiliary output
05H Printer output
06H Direct console I/O
07H Direct console input without echo
08H Console input without echo
09H Print string
0AH Buffered keyboard input
0BH Check standard input status
0CH Clear keyboard buffer, invoke a keyboard function
0DH Disk reset
0EH Select disk
0FH Open file
10H Close file
11H Search for first entry
12H Search for next entry
13H Delete file
14H Sequential read
15H Sequential write
16H Create file
17H Rename file
18H Reserved by DOS
19H Current disk
1AH Set disk transfer address
1BH Allocation table information
1CH Allocation table information for specific device
1DH Reserved by DOS
1EH Reserved by DOS
1FH Reserved by DOS
20H Reserved by DOS
21H Random read
22H Random write
23H File size
24H Set relative record field
25H Set interrupt vector
26H Create new program segment
27H Random block read

28H Random block write
29H Parse filename
2AH Get date
2BH Set date
2CH Get time
2DH Set time
2EH Set/reset verify switch
2FH Get disk transfer address
30H Get DOS version number
31H Terminate process and remain resident
32H Reserved by DOS
33H Ctrl-Break check
34H Reserved by DOS
35H Get vector
36H Get disk free space
37H Reserved by DOS
38H Set or get country dependent information
39H Create subdirectory (MKDIR)
3AH Remove subdirectory (RMDIR)
3BH Change current directory (CHDIR)
3CH Create a file (CREAT)
3DH Open a file
3EH Close a file handle
3FH Read from a file or device
40H Write to a file or device
41H Delete a file from a specified directory (UNLINK)
42H Move file read/write pointer (LSEEK)
43H Change file mode (CHMOD)
44H I/O control for devices (IOCTL)
45H Duplicate a file handle (DUP)
46H Force a duplicate of a file handle (FORCDUP)
47H Get current directory
48H Allocate memory
49H Free allocated memory
4AH Modify allocated memory blocks (SETBLOCK)
4BH Load or execute a program (EXEC)
4CH Terminate a process (EXIT)
4DH Get return code of a subprocess (WAIT)
4EH Find first matching file (FIND FIRST)
4FH Find next matching file
50H Reserved by DOS

51H Reserved by DOS
52H Reserved by DOS
53H Reserved by DOS
54H Get verify setting
55H Reserved by DOS
56H Rename a file
57H Get/set a file's date and time
58H Used internally by DOS
59H Get extended error
5AH Create Unique file
5BH Create new file
5CH Lock/unlock file access
5DH Reserved by DOS
5E00H Get machine name
5E02H Set printer setup
5E03H Get printer setup
5F02H Get redirection list entry
5F03H Redirect device
5F04H Cancel redirection
60H Reserved by DOS
61H Reserved by DOS
62H Get PSP address
63H Reserved by DOS
64H Reserved by DOS
65H Get Extended Country Information
66H Get/Set Global Code Page
67H Set Handle Count
68H Commit File

DOS Internal Stack

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 the user stack be 200H in addition to the user needs.

Error Return Information

Many of the function calls return the carry flag clear if the operation was successful. If an error condition was encountered, the carry flag is set.

If you are using DOS version 2.10, check the error code returned. For a list of error codes returned by function calls when you are using DOS 2.10, refer to "DOS 2.10 Error Codes" in this chapter.

If you are using DOS versions 3.00 to 3.30, use the Get Extended Error function call to return additional information about the error. For more information, refer to "Get Extended Error" in this chapter.

DOS 2.10 Error Codes

If you are using function calls 38H – 57H with DOS version 2.10, to check if an error has occurred, check for the following error codes in the AX register.

Function Call Number	Error Codes	Function Call	Error Codes
38H	2	44H	1,3,5,6
39H	3,5	45H	4,6
3AH	3,5,15	46H	4,6
3BH	3	47H	15
3CH	3,4,5	48H	7,8
3DH	2,3,4,5,12	49H	7,9
3EH	6	4AH	7,8,9
3FH	5,6	4BH	1,2,3,5,8,10,11
40H	5,6	4EH	2,3,18
41H	2,3,5	4FH	18
42H	1,6	56H	2,3,5,17
43H	1,2,3,5	57H	1,6

Get Extended Error (DOS 3.00 to 3.30)

The Get Extended Error function call (59H) is intended to provide a common set of error codes and to supply more extensive information about the error to the application. The information returned from function call 59H, in addition to the error code, is the error class, the locus, and the recommended action. The error class provides information about the error type (hardware, internal, system, etc.). The locus provides information about the area involved in the failure (serial device, block device, network, or memory). The recommended action provides a default action for programs that do not understand the specific error code.

Programs written from now on are expected to use the extended error support both from interrupt 24H hard error handlers and after any interrupt 21H function calls.

FCB function calls report an error by returning FFH in AL. Handle function calls report an error by setting the carry flag and returning the error code in AX. Interrupt 21H handle function calls for DOS 2.00 and 2.10 continue to return the error codes 1 – 18. Interrupt 24H handle function calls continue to return error codes 0 – 12. But the application can obtain any of the error codes listed in the extended error codes table by issuing function call 59H. Handle function calls, for DOS 3.00 to 3.30, can return any of the error codes. However, it is recommended that the function call is followed by function call 59H to obtain the error class, the locus, and the recommended action.

In order to create a common error table, error codes 0 – 12 from interrupt 24H correspond to error codes 19 – 31 in the extended error codes table. When a FAIL option is specified in the interrupt 24H error handler, issuing function call 59H returns error code 83 (FAIL on interrupt 24H).

The Extended Error Codes are grouped as follows:

0 No error
01 – 18 Error mappings for DOS 2.00/2.10 INT 21H errors
19 – 31 Error mappings for DOS 2.00/2.10 INT 24H errors
32 – 88 Errors for DOS 3.00 to 3.30

Note: Do not code to specific error codes. If you encounter an extended error code you do not recognize, perform the recommended action. Refer to "Actions" later in this chapter for more information.

Extended Error Codes

Many of the function calls return the carry flag clear if the operation was successful. If an error condition was encountered, the carry flag is set. To obtain information about the error, such as the *error class*, *locus*, and recommended *action*, issue the Get Extended Error function call 59H.

Code	Meaning
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
14	Reserved
15	Invalid drive was specified
16	Attempt to remove the current directory
17	Not same device
18	No more files
19	Attempt to write on write-protected diskette
20	Unknown unit
21	Drive not ready
22	Unknown command
23	Data error (CRC)
24	Bad request structure length
25	Seek error
26	Unknown media type
27	Sector not found
28	Printer out of paper
29	Write fault
30	Read fault
31	General failure
32	Sharing violation
33	Lock violation

34 Invalid disk change
35 FCB unavailable
36 Sharing buffer overflow
37–49 Reserved
50 Network request not supported
51 Remote computer not listening
52 Duplicate name on network
53 Network name not found
54 Network busy
55 Network device no longer exists
56 Net BIOS command limit exceeded
57 Network adapter hardware error
58 Incorrect response from network
59 Unexpected network error
60 Incompatible remote adapter
61 Print queue full
62 Not enough space for print file
63 Print file was deleted
64 Network name was deleted
65 Access denied
66 Network device type incorrect
67 Network name not found
68 Network name limit exceeded
69 Net BIOS session limit exceeded
70 Temporarily paused
71 Network request not accepted
72 Print or disk redirection is paused
73–79 Reserved
80 File exists
81 Reserved
82 Cannot make directory entry
83 Fail on INT 24
84 Too many redirections
85 Duplicate redirection
86 Invalid password
87 Invalid parameter
88 Network data fault

Error Classes

This value provides information about the type of error.

Value	Description
1	**Out of Resource:** Out of space, channels, etc.
2	**Temporary Situation:** Something that is expected to "go away" with time. Note that this is not an error condition, but a "situation" such as file locked, etc.
3	**Authorization:** Permission problem.
4	**Internal:** Internal error in system software. A situation judged to be a system software bug rather than a user or system failure.
5	**Hardware Failure:** A serious problem not the fault of user program.
6	**System Failure:** Serious failure of system software. Not directly the fault of the user. For example, configuration files missing or wrong.
7	**Application Program Error:** Inconsistent requests, etc.
8	**Not Found:** File/item not found.
9	**Bad Format:** File/item of invalid format, type, or otherwise invalid or unsuitable.
10	**Locked:** File/item interlocked.
11	**Media:** Media failure (wrong disk, CRC error...). Wrong disk in drive, bad spot on media, etc.
12	**Already Exists:** Collision with existing item, such as trying to declare a machine name that already exists.
13	**Unknown:** Classification doesn't exist or is inappropriate.

Actions

Note that these are recommended actions. In the most critical cases, the application will analyze the error codes and take specific action. These defaults are for programs that do not understand the specific error code.

Value	Description
1	**Retry:** Retry a few times, then prompt user to determine if the program should continue or be aborted.
2	**Delay Retry:** Retry after pause (a few times), then prompt user to determine if the program should continue or be aborted.
3	**User:** Ask user to reenter input. Typically, a bad drive letter or bad filename was presented in the system call. Naturally, if the value was "built into" the program and not directly keyed in by the user, then the program would not, in fact, "ask the user to reenter input." This action means that if the data came from a user, the best action is to tell him to try again.
4	**Abort:** Abort application with cleanup. The application cannot proceed, but the system is sufficiently healthy that the application should try an orderly shutdown.
5	**Immediate Exit:** Abort application immediately, skip cleanup. We do not recommend that the application try to close files, update indexes, but that it exit as soon as possible.
6	**Ignore:** Ignore.
7	**Retry After User Intervention:** The user needs to perform some action (like taking out a diskette and putting in a different one); then the operation should be retried.

Locus

This value provides additional information to help locate the area involved in the failure.

Value	Description
1	**Unknown:** Nonspecific. Not appropriate.
2	**Block Device:** Related to random access mass storage (disk).
3	**Net:** Related to the network.
4	**Serial Device:** Related to serial devices.
5	**Memory:** Related to random access memory.

ASCIIZ Strings

Several of the function 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.

The maximum size of an ASCIIZ string is 128 bytes, including the drive, colon, and null terminator.

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

Network Paths

For DOS 3.10 to 3.30, several of the function calls accept a network path as input if the IBM PC Local Area Network is loaded. A network path consists of an ASCII string containing a computer name, followed by a directory path, and in some cases a filename. The string cannot contain a drive specifier. The string is terminated by a byte of binary zeros. For example,

```
\\SERVER1\LEVEL1\LEVEL2\FILE1
```

All function calls that accept an ASCIIZ path as input, also accept a network path as input. Two function calls that do not accept a network path as input are Change Current Directory (3BH) and Find First Matching File (4EH).

Network Access Rights

The explanation of some function calls contains a section under remarks called "Network Access Rights." Any information under "Network Access Rights" tells you the access requirements for a directory that a computer on the network needs to be able to execute the function call when using DOS 3.10 to 3.30. For example, suppose you want to execute function call 5BH (Create New File). You must have Read/Write/Create or Write/Create access to the directory to be able to create a file. If you have Read Only or Write Only access (no Create access), you cannot create a file in the directory.

File Handles

The extended function calls (3CH-62H) that supporting files or devices use an identifier known as a "handle." When you create or open a file or device with these 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 predefined by DOS and can be used by your program. You do not need to open them before using them:

Hex Value Meaning

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.

Using 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 used in one of these ways:

1. Place the function number in AH and execute a long call to offset 50H in your program segment prefix.

2. Place the function number in AH and issue interrupt type 21H. This is the preferred method of using DOS function calls.

3. There is an additional mechanism provided for preexisting 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 (00H-24H).

Notes:

1. All FCB function calls do not allow invalid characters (0DH-29H).

2. Device names cannot end in a colon (:).

3. The contents of the AX register may be altered by any of the function calls. Even if no error code is returned in AX, the user cannot be guaranteed that AX is unchanged.

4. Function calls 01H through 0CH use the standard devices listed in the "File Handles" section. Refer to "File Handles" on page 6-48 for more information.

00H
Program Terminate

Purpose:

Terminates the execution of a program.

On Entry	Register Contents
AH	00H
CS	Points to PSP

On Return	Register Contents
	NONE

Remarks:

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 and the handles opened by the process are closed. Any files that have changed in length and not closed are not recorded properly in the directory. Control transfers to the terminate address. This call performs exactly the same function as interrupt 20H. It is the program's responsibility to ensure that the CS register contains the segment address of its program segment prefix control block before calling this function.

01H
Keyboard Input

Purpose:

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 the character in AL.

On Entry	Register Contents
AH	01H

On Return	Register Contents
AL	Character from the standard input device

Remarks:

The character is checked for a Ctrl – Break. If Ctrl – Break is detected, an interrupt 23H is executed.

> **Note:** For function call 01H, extended ASCII codes require two function calls. The first call returns 00H as an indicator that the next call will return an extended code. Refer to "Extended ASCII Codes" in the beginning of this chapter for a table of Extended ASCII codes.

02H
Display Output

Purpose:

Outputs the character in DL to the standard output device.

On Entry	Register Contents
AH	02H
DL	Character

On Return	Register Contents
	NONE

Remarks:

If the character in DL is a backspace (08), the cursor is moved left on position (nondestructive). If a Ctrl – Break is detected after the output, an interrupt 23H is executed.

03H
Auxiliary Input

Purpose:

Waits for a character from the standard auxiliary device, then returns that character in AL.

On Entry	Register Contents
AH	03H

On Return	Register Contents
AL	Character from the auxiliary device

Remarks:

Auxiliary (AUX, COM1, COM2, COM3 and COM4) support is unbuffered and noninterrupt driven.

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

The auxiliary function calls (03H and 04H) do not return status or error codes. For greater control, it is recommended that you use the ROM BIOS routine (interrupt 14H) or write an AUX device drivers and use IOCTL.

04H
Auxiliary Output

Purpose:

Outputs the character in DL to the standard auxiliary device.

On Entry	Register Contents
AH	04H
DL	Character

On Return	Register Contents
	NONE

05H
Printer Output

Purpose:

Outputs the character in DL to the standard printer device.

On Entry	Register Contents
AH	05H
DL	Character

On Return	Register Contents
	NONE

06H
Direct Console I O

Purpose:

Waits for a character from the standard input device if one is ready.

On Entry	Register Contents
AH	06H
DL	FFH, for console input 00H-FEH, for console output

On Return	Register Contents
AL	See description below

Remarks:

If DL is FFH, 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 FFH, 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.

> **Note:** For function call 06H, extended ASCII codes require two function calls. The first call returns 00H as an indicator that the next call will return an extended code. Refer to "Extended ASCII Codes" in the beginning of

this chapter for a table of Extended ASCII codes.

07H
Direct Console Input Without Echo

Purpose:

Waits for a character to be read at the standard input device (unless one is ready), then returns the character in AL.

On Entry	Register Contents
AH	07H

On Return	Register Contents
AL	Character from standard input device

Remarks:

As with function call 06H, no checks are made on the character.

08H
Console Input Without Echo

Purpose:

Waits for a character to be read at the standard input device (unless one is ready) and returns the character in AL.

On Entry	Register Contents
AH	08H

On Return	Register Contents
AL	Character from standard input device

Remarks:

The character is checked for Ctrl – Break. If Ctrl – Break is detected, an interrupt 23H is executed.

Note: For function call 08H, extended ASCII codes require two function calls. The first call returns 00H as an indicator that the next call will return an extended code. Refer to "Extended ASCII Codes" in the beginning of this chapter for a table of Extended ASCII codes.

09H
Print String

Purpose:

Outputs the characters in the print string to the standard output device.

On Entry	Register Contents
AH	09H
DS:DX	Pointer to the character string

On Return	Register Contents
	NONE

Remarks:

The character string in memory must be terminated by a **$** (24H). Each character in the string is output to the standard output device in the same form as function call 02H.

0AH
Buffered Keyboard Input

Purpose:

Reads characters from the standard input device and places them in the buffer beginning at the third byte.

On Entry	Register Contents
AH	0AH
DS:DX	Pointer to an input buffer

On Return	Register Contents
	NONE

Remarks:

The first byte of the input buffer specifies the number of characters the buffer can hold. This value cannot be zero. 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, 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 (0DH), which is always the last character.

0BH
Check Standard Input Status

Purpose:

Checks if there is a character available from the standard input device.

On Entry	Register Contents
AH	0BH

On Return	Register Contents
AL	FFH If the character is available from the standard input device 00H If no character is available from the standard input device

Remarks:

If a character is available from the standard input device, AL is FFH. Otherwise, AL is 00H. If a Ctrl – Break is detected, an interrupt 23H is executed.

0CH
Clear Keyboard Buffer and Invoke a Keyboard Function

Purpose:

Clears the standard input buffer of any pretyped characters, then executes the function call number in AL (only 01H, 06H, 07H, 08H, and 0AH are allowed).

On Entry	Register Contents
AH	0CH
AL	Function number

On Return	Register Contents
	NONE

Remarks:

This forces the system to wait until a character is typed.

0DH
Disk Reset

Purpose:

Writes file buffers that have been modified to the disk.

On Entry	Register Contents
AH	0DH

On Return	Register Contents
	NONE

Remarks:

It is still necessary to close all open files to correctly update the disk directory.

0EH
Select Disk

Purpose:

Selects the drive specified in DL (0 = A, 1 = B, etc.) (if valid) as the default drive.

On Entry	Register Contents
AH	0EH
DL	Drive number (0 = A, 1 = B, etc.)

On Return	Register Contents
AL	Total number of drives

Remarks:

The number of unique drive letters that can be referenced (total of diskette and fixed disk drives) is returned in AL. The value in AL is equal to the value of LASTDRIVE in CONFIG.SYS or the total number of installed devices, whichever is greater. For DOS 3.00 to 3.30, the minimum value returned in AL is 5. If the system has only one diskette drive, it is counted as two to be consistent with the philosophy of thinking of the system as having logical drives A and B.

0FH
Open File

Purpose:

Searches the current directory for the named file and AL returns FFH if it is not found. If it is found, AL returns 00H and the FCB is filled as described below.

On Entry	Register Contents
AH	0FH
DS:DX	Pointer to an unopened FCB

On Return	Register Contents
AL	00H If file opened FFH If file not opened

Remarks:

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 80H. The size of the file and the date are set in the FCB from information obtained from the directory. You can change the default value for the record size (FCB bytes E-F) or set the random record size and/or current record field. Perform these actions after the open but before any disk operations.

0FH
Open File

The file is opened in compatibility mode. For information on compatibility mode, refer to function call 3DH in this chapter.

10H
Close File

Purpose:

Closes a file after a file write.

On Entry	Register Contents
AH	10H
DS:DX	Pointer to an opened FCB

On Return	Register Contents
AL	00H If the file is found FFH If the file is not found in the current directory

Remarks:

This function call must be done on open files that are no longer needed, and after file writes to ensure all directory information is updated. If the file is not found in its correct position in the current directory, it is assumed the diskette was changed and AL returns FFH. Otherwise, the directory is updated to reflect the status in the FCB, the buffers for that file are flushed, and AL returns 00H.

11H
Search for First Entry

Purpose:

Searches for the first matching filename.

On Entry	Register Contents
AH	11H
DS:DX	Pointer to an unopened FCB

On Return	Register Contents
AL	00H If matching filename found FFH If matching filename was not found

Remarks:

The current disk directory is searched for the first matching filename. If none are found, AL returns FFH. For DOS 2.10, question marks (?)s are allowed in the filename. For DOS 3.00 to 3.30, global filename characters are allowed. If a matching filename is found, AL returns 00H 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 FFH followed by 5 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 standard 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, are not 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 "DOS Disk Directory" on page 5-10.

12H
Search for Next Entry

Purpose:

Searches the current directory for the next matching filename.

On Entry	Register Contents
AH	12H
DS:DX	Pointer to an the unopened FCB specified from the previous Search First (11H) or Search Next (12H).

On Return	Register Contents
AL	00H If matching filename found FFH If matching filename not found

Remarks:

After a matching filename has been found using function call 11H, function 12H may be called to find the next match to an ambiguous request. For DOS 2.10, question marks (?)s are allowed in the filename. For DOS 3.00 to 3.30, global filename characters are allowed.

The DTA contains information from the previous Search First or Search Next. All of the FCB except for the name/extension field is used to keep information necessary for continuing the search, so

between a previous function 11H or 12H call and this one.

13H
Delete File

Purpose:

Deletes all current directory entries that match the specified filename. The specified filename cannot be read – only.

On Entry	Register Contents
AH	13H
DS:DX	Pointer to an unopened FCB

On Return	Register Contents
AL	00H File deleted FFH If directory entry match was not found

Remarks:

All matching current directory entries are deleted. The global filename character "?" is allowed in the filename. If no directory entries match, AL returns FFH; otherwise AL returns 00H.

If the file is specified in read – only mode, the file is not deleted.

Note: Close open files before deleting them.

Network Access Rights: Requires Create access rights.

14H
Sequential Read

Purpose:

Loads the record addressed by the current block (FCB bytes C-D) and the current record (FCB byte 1F) at the disk transfer address (DTA), then the record address is incremented.

On Entry	Register Contents
AH	14H
DS:DX	Pointer to an opened FCB

On Return	Register Contents
AL	00H If read was successfully completed 01H If EOF (no data read) 02H If the read would have caused a wrap or overflow because the DTA was too small (The read was not completed.) 03H If EOF (a partial record was read and filled out with zeros)

Remarks:

The length of the record is determined by the FCB record size field.

Network Access Rights: Requires Read access rights.

15H
Sequential Write

Purpose:

Writes the record addressed by the current block and record fields (size determined by the FCB record size field) from the disk transfer address. If records are less than the sector size, the record is buffered for an eventual write when a sector's worth of data is accumulated. Then the record address is incremented.

On Entry	Register Contents
AH	15H
DS:DX	Pointer to an opened FCB

On Return	Register Contents
AL	00H If write was successfully completed 01H If diskette is full (write canceled) 02H If DTA too small (write canceled)

Remarks:

If the file is specified in read-only mode, the sequential write is not performed.

Network Access Rights: Requires Write access rights.

16H
Create File

Purpose:

Searches the current directory of the specified drive for a matching entry.

On Entry	Register Contents
AH	16H
DS:DX	Pointer to an unopened FCB

On Return	Register Contents
AL	00H If file created (matching entry found or empty entry found) FFH If file not created (full directory or disk and no matching directory entry)

Remarks:

If a matching entry is found it is reused. If no match is found, the directory is searched for an empty entry. If a match is found, the entry is initialized to a zero-length file, the file is opened (see function call 0FH), and AL returns 00H.

16H
Create File

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

Network Access Rights: Requires Create access rights.

17H
Rename File

Purpose:

Changes every matching occurrence of the first filename in the current directory of the specified drive to the second (with the restriction that two files cannot have the same name and extension.)

On Entry	Register Contents
AH	17H
DS:DX	Pointer to a modified FCB

On Return	Register Contents
AL	00H If file renamed (matching filename found) FFH If no matching filename found or if an attempt to rename an existing filename

Remarks:

The modified FCB has a drive code and filename in the usual position, and a second filename starting 6 bytes after the first (DS:DX + 11H) in what is normally a reserved area. If "?"s appear in the second name, then the corresponding positions in the original name are unchanged.

17H
Rename File

If the file is specified in read-only mode, the file is not renamed.

Network Access Rights: Requires Create access rights.

19H
Current Disk

Purpose:

Determines the current default drive.

On Entry	Register Contents
AH	19H

On Return	Register Contents
AL	Current default drive (0 = A, 1 = B, etc.)

Remarks:

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

1AH
Set Disk Transfer Address

Purpose:

Sets the disk transfer address to DS:DX.

On Entry	Register Contents
AH	1AH
DS:DX	Disk transfer address

On Return	Register Contents
	NONE

Remarks:

The area defined by this call is from the address in DS:DX to the end of the segment in DS. DOS does not allow disk transfers to wrap around within the segment, or overflow into the next segment. If you do not set the DTA, the default DTA is offset 80H in the program segment prefix.

> **Note:** You can get the DTA using function call 2FH.

1BH
Allocation Table Information

Purpose:

Returns information about the allocation table for the default drive.

On Entry	Register Contents
AH	1BH

On Return	Register Contents
DS:BX	Pointer to the media descriptor byte for the default drive
DX	Number of allocation units
AL	Number of sectors/allocation unit
CX	Size of the physical sector

Remarks:

For more information on DOS disk allocation, refer to "DOS Disk Directory" on page 5-10. Also, refer to function call 36H (Get Disk Free Space).

1CH
Allocation Table Information for Specific Device

Purpose:

Returns allocation table information for a specific device.

On Entry	Register Contents
AH	1CH
DL	Drive number

On Return	Register Contents
DS:BX	Points to the media descriptor byte of the drive specified in DL
AL	Number of sectors/allocation unit
DX	Number of allocation units
CX	Size of the physical sector

Remarks:

This call is identical to call 1BH except that, on entry, DL contains the number of the drive that contains the needed information (0 = default, 1 = A, etc.). For more information on DOS disk allocation, refer to "DOS Disk Directory" on page 5-10. Also, refer to function call 36H (Get Disk Free Space).

21H
Random Read

Purpose:

Reads the record addressed by the current block and current record fields into memory at the current disk transfer address.

On Entry	Register Contents
AH	21H
DS:DX	Pointer to an opened FCB

On Return	Register Contents
AL	00H If read was successfully completed 01H If EOF (no data read) 02H If the read would have caused a wrap or overflow because the DTA was too small (The read was not completed.) 03H If EOF (a partial record was read and filled out with zeros)

Remarks:

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.

Network Access Rights: Requires Read access rights.

22H
Random Write

Purpose:

Writes the record addressed by the current block and current record fields from the current disk transfer address.

On Entry	Register Contents
AH	22H
DS:DX	Pointer to an opened FCB

On Return	Register Contents
AL	00H If write was successfully completed 01H If diskette is full (write canceled) 02H If DTA too small (write canceled)

Remarks:

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 file is specified in read – only mode, the random write is not performed.

Network Access Rights: Requires Write access rights.

23H
File Size

Purpose:

Searches the diskette directory for an entry that matches the specified file and sets the FCBs random record field to the number of records in the file.

On Entry	Register Contents
AH	23H
DS:DX	Pointer to an unopened FCB

On Return	Register Contents
AL	00H If the directory entry is found FFH If the directory entry not found

Remarks:

The diskette directory is searched for the matching entry. If a matching entry is found, the random record field is set to the number of records in the file (in terms of the record size field rounded up). If no matching entry is found, AL returns FFH.

> **Note:** If you do not set the FCB record size field before using this function, incorrect information is returned.

24H
Set Relative Record Field

Purpose:

Sets the random record field to the same file address as the current block and record fields.

On Entry	Register Contents
AH	24H
DS:DX	Pointer to an opened FCB

On Return	Register Contents
	NONE

Remarks:

You must call this function before you perform random read and writes, and random block read and writes.

25H
Set Interrupt Vector

Purpose:

Sets the interrupt vector table for the interrupt number.

On Entry	Register Contents
AH	25H
DS:DX	Address of interrupt handling routine
AL	Interrupt number

On Return	Register Contents
	NONE

Remarks:

The interrupt vector table for the interrupt number specified in AL is set to address contained in DS:DX. Use function call 35H (Get Vector) to obtain the contents of the interrupt vector.

26H
Create New Program Segment

Purpose:

Creates a new program segment.

On Entry	Register Contents
AH	26H
DX	Segment number for the new program segment

On Return	Register Contents
	NONE

Remarks:

The entire 100H area at location 0 in the current program segment is copied into location 0 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 interrupts 22H, 23H, and 24H are saved in the new program segment starting at 0AH. They are restored from this area when the program terminates.

> **Note:** You should avoid using this call. We recommend that you use the EXEC function call 4BH instead.

27H
Random Block Read

Purpose:

Reads the specified number of records (in terms of the record size field) from the file address specified by the random record field into the disk transfer address.

On Entry	Register Contents
AH	27H
DS:DX	Pointer to an opened FCB
CX	Number of records to be read

On Return	Register Contents
AL	00H If read was successfully completed 01H If EOF (no data read) 02H If the read would have caused a wrap or overflow because the DTA was too small (The read was not completed.) 03H If EOF (a partial record was read and filled out with zeros)
CX	Actual number of records read

27H
Random Block Read

Remarks:

The random record field and the current block/record fields are set to address the next record (the first record not read).

Network Access Rights: Requires Read access rights.

28H
Random Block Write

Purpose:

Writes the specified number of records from the disk transfer address into the file address specified by the random record field.

On Entry	Register Contents
AH	28H
DS:DX	Pointer to an opened FCB
CX	Number of records to be written

On Return	Register Contents
AL	00H If write was successfully completed 01H If diskette is full (write canceled) 02H If DTA too small (write canceled)
CX	Actual number of records written

Remarks:

If there is insufficient space on the disk, AL returns 01H 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.)

28H
Random Block Write

Network Access Rights: Requires Write access rights.

29H
Parse Filename

Purpose:

Parses the specified filename.

On Entry	Register Contents
AH	29H
DS:SI	Pointer to a command line to parse
ES:DI	Pointer to a portion of memory that will be filled with an unopened FCB
AL	Bit value controls parsing

On Return	Register Contents
AL	00H If no global filename characters in command line 01H If global filename characters used in command line FFH If drive specifier invalid
DS:SI	Points to the first character after the parsed filename
ES:DI	Points to the first byte of the formatted FCB

29H
Parse Filename

Remarks:

The contents of AL are used to determine the action to take, as shown below:

```
        <must = 0>
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, 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 01H, if the drive specifier is invalid, AL returns FFH, otherwise 00H.

29H
Parse Filename

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

2AH
Get Date

Purpose:

Returns the day of the week, year, month and date.

On Entry	Register Contents
AH	2AH

On Return	Register Contents
AL	Day of the week (0=SUN 6=SAT)
CX	Year (1980 - 2099)
DH	Month (1 - 12)
DL	Day (1 - 31)

Remarks:

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.

2BH
Set Date

Purpose:

Sets the date (also sets CMOS clock, if present).

On Entry	Register Contents
AH	2BH
CX	Year (1980 - 2099)
DH	Month (1 - 12)
DL	Day (1 - 31)

On Return	Register Contents
AL	00H, if the date is valid FFH, if the date is not valid

Remarks:

On entry, CX:DX must have a valid date in the same format as returned by function call 2AH.

On return, AL returns 00H if the date is valid and the set operation is successful. AL returns FFH if the date is not valid.

2CH
Get Time

Purpose:

Returns the time; hours, minutes, seconds and hundredths of seconds.

On Entry	Register Contents
AH	2CH

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

Remarks:

On entry, AH contains 2CH. On return, CX:DX contains the time – of – day. 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.

2DH
Set Time

Purpose:

Sets the time (also sets the CMOS clock, if present).

On Entry	Register Contents
AH	2DH
CH	Hour (0 -23)
DH	Seconds (0 - 59)
CL	Minutes (0 - 59)
DL	Hundredths (0 - 99)

On Return	Register Contents
AL	00H, if the time is valid FFH, if the time is not valid

Remarks:

On entry, CX:DX has time in the same format as returned by function 2CH. On return, if any component of the time is not valid, the set operation is aborted and AL returns FFH. If the time is valid, AL returns 00H.

If your system has a CMOS realtime clock, it will be set.

2EH
Set/Reset Verify Switch

Purpose:

Sets the verify switch.

On Entry	Register Contents
AH	2EH
AL	00H, to set verify off 01H, to set verify on

On Return	Register Contents
	NONE

Remarks:

On entry, AL must contain 01H to turn verify on, or 00H to turn verify off. When verify is on, DOS performs a verify operation each time it performs a disk write to assure proper data recording. Although disk recording errors are very rare, this function has been provided for applications in which you may wish to verify the proper recording of critical data. You can obtain the current setting of the verify switch through function call 54H.

Note: Verification is not supported on data written to a network disk.

2FH
Get Disk Transfer Address (DTA)

Purpose:

Returns the current disk transfer address.

On Entry	Register Contents
AH	2FH

On Return	Register Contents
ES:BX	The current DTA

Remarks:

On entry, AH contains 2FH. On return, ES:BX contains the current Disk Transfer Address. You can set the DTA using function call 1AH.

30H
Get DOS Version Number

Purpose:

Returns the DOS version number.

On Entry	Register Contents
AH	30H

On Return	Register Contents
BX	0000H
CX	0000H
AL	Major version number
AH	Minor version number

Remarks:

On entry, AH contains 30H. On return, BX and CX are set to 0. AL contains the major version number. AH contains the minor version number. For example, for DOS 3.10., the major version number is 03H and the minor version number is 0AH. For DOS 3.20 the major version number is 03H and the minor version number is 14H.

> **Note:** If AL returns a major version number of zero, then it can be assumed that the DOS version is pre−DOS 2.00.

31H
Terminate Process and Remain Resident

Purpose:

Terminates the current process and attempts to set the initial allocation block to the memory size in paragraphs.

On Entry	Register Contents
AH	31H
AL	Return code
DX	Memory size in paragraphs

On Return	Register Contents
	NONE

Remarks:

On entry, AL contains a binary return code. DX contains the memory size value in paragraphs. This function call does not free up any other allocation blocks belonging to that process. Files opened by the process are not closed when the call is executed. The return code passed in AL is retrievable by the parent through Wait (function call 4DH) and can be tested through the ERRORLEVEL batch subcommands.

> **Note:** Memory can be more efficiently used if the block containing a copy of the environment is deallocated before terminating. This can be done by loading ES with the segment

31H
Terminate Process and Remain Resident

contained in 2C of the PSP, and issuing function call 49H (Free Allocated Memory).

33H
Ctrl-Break Check

Purpose:

Set or get the state of BREAK (Ctrl – Break checking).

On Entry	Register Contents
AH	33H
AL	00H, to request current state 01H, to set the current state
DL	00H, to set current state OFF 01H, to set current state ON

On Return	Register Contents
DL	The current state (00H = OFF, 01H = ON)

Remarks:

On entry, AL contains 00H to request the current state of Ctrl-Break checking, 01H to set the state. If setting the state, DL must contain 00H for OFF or 01H for ON. On return, if requesting the current state, DL contains the current state (00H = OFF, 01H = ON).

35H
Get Vector

Purpose:

To obtain the address in an interrupt vector.

On Entry	Register Contents
AH	35H
AL	Interrupt number

On Return	Register Contents
ES:BX	Pointer to the interrupt handling routine.

Remarks:

On entry, AH contains 35H. AL contains a hexadecimal interrupt number. On return, ES:BX contains the CS:IP interrupt vector for the specified interrupt. Use function call 25H (Set Interrupt Vector) to set the interrupt vectors.

36H
Get Disk Free Space

Purpose:

Returns the disk free space (available clusters, clusters/drive, bytes/sector).

On Entry	Register Contents
AH	36H
DL	Drive (0 = default, 1 = A)

On Return	Register Contents
BX	Available clusters
DX	Clusters/drive
CX	Bytes/sector
AX	FFFFH if the drive in DL is invalid, otherwise the number of sectors per cluster

Remarks:

If the drive number in DL was valid, 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 (1BH).

38H (DOS 2.10)
Return Country Dependent Information

Purpose:

Returns country dependent information.

On Entry	Register Contents
AH	38H
DS:DX	Pointer to the 32-byte memory area
AL	Equals the function code

On Return	Register Contents
AX	Error code if carry flag set
DS:DX	Country data if carry flag not set

Remarks:

On entry, DS:DX points to a 32-byte block of memory in which returned information is passed and AL contains a function code. In DOS 2.10, this function code must be 0. The following information is pertinent to international applications:

WORD date/time format
BYTE ASCIIZ string currency symbol followed by byte of zeros

38H (DOS 2.10)
Return Country Dependent Information

BYTE ASCIIZ string thousands separator followed by byte of zeros
BYTE ASCIIZ string decimal separator followed by byte of zeros
24 bytes Reserved

The time and date 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 y:m:d

38H (DOS 3.00 to 3.30)
Get or Set Country Dependent Information

Purpose:

Returns country dependent information.

Get Current Country

On Entry	Register Contents
AH	38H
DS:DX	Pointer to the memory buffer where the data will be returned
AL	00H ; to get current country information Country code ; to get information for countries with a code <255 FFH ; to get country information for countries with a code ≥ 255
BX	16 bit country code ; if AL = FFH

38H (DOS 3.00 to 3.30)
Get or Set Country Dependent Information

On Return	Register Contents
AX	Error code if carry flag set
DS:DX	Filled with the country information
BX	Country code

38H (DOS 3.00 to 3.30)
Get or Set Country Dependent Information

Set Current Country

On Entry	Register Contents
AH	38H
DS:DX	FFFFH
AL	Country code for countries with a code <255 FFH for countries with a code ≥ 255
BX	16-bit country code ; if AL=FFH

On Return	Register Contents
AX	Error code if carry flag set

38H (DOS 3.00 to 3.30)
Get or Set Country Dependent Information

Country Information

WORD Binary value indicating the date format
5 BYTE currency symbol null terminated
2 BYTE thousands separator null terminated
2 BYTE decimal separator null terminated
2 BYTE date separator null terminated
2 BYTE time separator null terminated
1 BYTE binary value indicating the currency format 0 = the currency symbol precedes the value, no spaces between the symbol and value 1 = the currency symbol follows the value, no spaces between the symbol and value 2 = the currency symbol precedes the value, one space between the symbol and value 3 = the currency symbol follows the value, one space between the symbol and value 4 = the currency symbol replaces the decimal separator
1 BYTE number of significant decimal digits in currency
1 BYTE time format Bit 0 = 0 if 12−hour clock Bit 0 = 1 if 24−hour clock
2 WORDS Case map call address for codes > 7FH
2 BYTES Data list separator null terminated
5 WORDS Reserved

38H (DOS 3.00 to 3.30)
Get or Set Country Dependent Information

The date format has the following values and meaning:

Code	Date
0 = USA	m d y
1 = Europe	d m y
2 = Japan	y m d

Case Map Call Address: The register contents for the case map call are:

On Entry	Register Contents
AL	ASCII code of character to be converted to uppercase

On Return	Register Contents
AL	ASCII code of the uppercase input character

The case map call address is in a form suitable for a FAR call indirect.

38H (DOS 3.00 to 3.30) Get or Set Country Dependent Information

Remarks:

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Keyboard considerations:

For country codes other than 001 and 061, an alternate keyboard handler should have been invoked in the AUTOEXEC.BAT file.

When an alternate keyboard handler is invoked, the keyboard routine is loaded into user memory starting at the lowest portion of available user memory. The BIOS interrupt vector that services the keyboard is changed by the routine to redirect the CPU to the section of user memory where the new keyboard routine now resides. Each keyboard routine has lookup tables that return ASCII values unique to each language. Refer to the KEYB command in the *DOS Reference*.

Once the keyboard interrupt vector is changed by the DOS keyboard routine, the interrupt is always serviced by the routine in read/write memory. Return to the U.S. English keyboard format is available by holding the Ctrl and Alt keys and pressing F1 at the same time. This does not change the interrrupt vector back to the BIOS location. In this case, the interrupt is still processed by the read/write routine, but the routine looks up the scan codes, which will be converted to ASCII codes, in the same manner as ROM BIOS. However, Ctrl – Alt – F1 does not return you to the U.S.

keyboard if you are using a computer with ROM keyboard support. Similarly, holding the Ctrl and Alt keys and pressing F2 causes a return to the read/write lookup tables.

39H
Create Subdirectory (MKDIR)

Purpose:

Creates the specified directory.

On Entry	Register Contents
AH	39H
DS:DX	Pointer to an ASCIIZ string

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

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 created. On return, a new directory is created at the end of the specified path.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: Requires Create access rights.

3AH
Remove Subdirectory (RMDIR)

Purpose:

Removes the specified directory.

On Entry	Register Contents
AH	3AH
DS:DX	Pointer to an ASCIIZ string

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

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 codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: Requires Create access rights.

3BH
Change the Current Directory (CHDIR)

Purpose:

Changes the current directory to the specified directory.

On Entry	Register Contents
AH	3BH
DS:DX	Pointer to an ASCIIZ string

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

On entry, DS:DX contains the address of an ASCIIZ string with drive and directory path names. The string is limited to 64 characters and cannot contain a network path. 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 codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

3CH
Create a File (CREAT)

Purpose:

Creates a new file or truncates an old file to zero length in preparation for writing.

On Entry	Register Contents
AH	3CH
DS:DX	Pointer to an ASCIIZ string
CX	Attribute of the file

On Return	Register Contents
AX	Error codes if carry flag is set 16-bit handle if carry flag not set

Remarks:

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, the read/write pointer is set to the first byte of the file and the handle is returned in AX. Note that the change mode function call (43H) can later be used to change the file's attribute.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: Requires Create access rights.

3DH (DOS 2.10)
Open a File

Purpose:

Opens the specified file.

On Entry	Register Contents
AH	3DH
DS:DX	Pointer to an ASCIIZ path name
AL	Access Code

On Return	Register Contents
AX	Error codes if carry flag is set 16 – bit file handle if carry flag not set

Remarks:

This call opens any normal or hidden file whose name matches the name specified. Files that end with a colon are not opened.

The read/write pointer is set at the first byte of the file and the record size of the file is 1 byte (the read/write pointer can be changed through function call 42H). 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 57H, and its attribute can be obtained through call 43H.

3DH (DOS 2.10)
Open a File

Access Codes

AL = 0 File is opened for reading

AL = 1 File is opened for writing

AL = 2 File is opened for both reading and writing

3DH (DOS 3.00 to 3.30)
Open a File

Purpose:

Opens the specified file.

On Entry	Register Contents
AH	3DH
DS:DX	Pointer to an ASCIIZ path name
AL	Open mode

On Return	Register Contents
AX	Error codes if carry flag is set 16-bit file handle if carry flag not set

Remarks:

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 42H). 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 57H, and its attribute can be obtained through call 43H.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on

page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: If the Access field (A) of the Open mode field (AL) is equal to:

000 Requires Read access rights

001 Requires Write access rights

010 Requires Read/Write access rights

Notes:

1. This call opens any normal or hidden file whose name matches the name specified. Files that end with a colon are not opened.

2. When a file is closed, any sharing restrictions placed on it by the open are canceled.

3. File sharing must be loaded for the sharing modes to function. Refer to the SHARE command in Chapter 7 "DOS Commands" of the *DOS Reference*.

4. The file read – only attribute can be set when creating the file using extended FCBs or specifying the appropriate attribute in CX for the handle creates by using the CHMOD interrupt 21 function call or the DOS ATTRIB command.

5. If the file is inherited by the child process, all sharing and access restrictions are also inherited.

6. If an open file handle is duplicated by either of the DUP function calls, all sharing and access restrictions are also duplicated.

3DH (DOS 3.00 to 3.30)
Open a File

Open Mode

The open mode is defined in AL and consists of four bit – oriented fields. They are the:

- Inheritance flag
- Sharing mode field
- Reserved field
- Access field

The inheritance flag specifies if the opened file will be inherited by a child process. The access field defines what operations this process may perform on the file. The sharing mode field defines what operations other processes may perform on the file.

Bit Fields

The bit fields are mapped as follows:

```
                  <I> < S > <R> < A >
Open Mode bits     7  6 5 4  3  2 1 0
```

I Inheritance flag

If I = 0; File is inherited by child processes
If I = 1; File is private to the current process

S Sharing Mode

The file is opened as follows:

If S = 000; Compatibility mode
If S = 001; Deny Read/Write mode (Exclusive)
If S = 010; Deny Write mode
If S = 011; Deny Read mode
If S = 100; Deny None mode

Any other combinations are invalid.

When opening a file, it is important to inform DOS what operations other processes may perform on this file (sharing mode). The default (compatibility mode) denies all other computers on a network access to the file. Perhaps it is all right for other processes to continue to read this file while your process is operating on the file. In this case, you should specify Deny/Write, which inhibits writing by other processes, but allows reading by them.

Similarly, it is important to specify what operations your process will perform (access mode). The default access mode (Read/Write) causes the open request to fail if another process on this computer or any other computer on a network has the file opened with any sharing mode other than deny none. If however, all you intended to do is read from the file, your open will succeed unless all other processes have specified deny none or deny write (therefore increasing access to the file). File sharing requires cooperation of both sharing processes. This cooperation is communicated through the sharing and access mode.

R Reserved (set third bit field to 0).

A Access

The file access is assigned as follows:

If A = 000; Read access
If A = 001; Write access
If A = 010; Read/Write access

Any other combinations are invalid.

Compatibility Mode

A file is considered to be in compatibility mode if the file is opened by:

- Any of the CREATE function calls
- An FCB function call
- A handle function call with compatibility mode specified

A file can be opened any number of times in compatibility mode by a single process, provided that the file is not currently open under one of the other four sharing modes. If the file is marked read-only, and is currently open in Deny Write sharing mode with Read Access, the file may be opened in Compatibility Mode with Read Access. If the file was successfully opened in one of the other sharing modes and an attempt is made to open the file again in Compatibility Mode, an interrupt 24H is generated to signal this error. The base interrupt 24H error indicates **Drive not ready**, and the extended error indicates a **Sharing violation**.

Sharing Modes

The sharing modes for a file opened in compatibility mode are changed by DOS depending on the read-only attribute of the file. This is to allow sharing of read-only files.

3DH (DOS 3.00 to 3.30)
Open a File

File Opened By	Read – Only Access	Sharing Mode
FCB	Read-Only	Deny Write
Handle Read	Read-Only	Deny Write
Handle Write	Error	-----
Handle Read/Write	Error	-----

File Opened By	Not Read – Only Access	Sharing Mode
FCB	Read/Write	Compatibility
Handle Read	Read	Compatibility
Handle Write	Write	Compatibility
Handle Read/Write	Read/Write	Compatibility

Deny Read/Write Mode (Exclusive)

If a file is successfully opened in Deny Read/Write mode, access to the file is exclusive. A file currently open in this mode cannot be opened again in any sharing mode by any process (including the current process) until the file is closed.

Deny Write Mode

A file successfully opened in Deny Write sharing mode, prevents any other write access opens to the file (A = 001 or 010) until the file is closed. An attempt to open a file in Deny Write mode is unsuccessful if the file is currently open with a write access.

Deny Read Mode

A file successfully opened in Deny Read sharing mode, prevents any other read sharing access opens to the file (A = 000 or 010) until the file is closed. An attempt to open a file in Deny Read sharing mode is unsuccessful if the file is currently open in Compatibility mode or with a read access.

3DH (DOS 3.00 to 3.30)
Open a File

Deny None Mode

A file successfully opened in Deny None mode, places no restrictions on the read/write accessibility of the file. An attempt to open a file in Deny None mode is unsuccessful if the file is currently open in Compatibility mode.

Note: When accessing files that reside on a network disk, no local buffering is done when when files are opened in any of the following sharing modes:

- Deny Read
- Deny None

Therefore, in a network environment, Deny Read/Write sharing mode, Compatibility sharing mode, and Deny Write mode opens are buffered locally.

The following sharing matrix shows the results of opening, and subsequently attempting to reopen the same file using all combinations of access and sharing modes.

3DH (DOS 3.00 to 3.30)
Open a File

		DRW			DW			DR			ALL		
		I	IO	O	I	IO	O	I	IO	O	I	IO	O
DRW	I	N	N	N	N	N	N	N	N	N	N	N	N
	IO	N	N	N	N	N	N	N	N	N	N	N	N
	O	N	N	N	N	N	N	N	N	N	N	N	N
DW	I	N	N	N	Y	N	N	N	N	N	Y	N	N
	IO	N	N	N	N	N	N	N	N	N	Y	N	N
	O	N	N	N	N	N	N	Y	N	N	Y	N	N
DR	I	N	N	N	N	N	N	N	N	N	N	N	Y
	IO	N	N	N	N	N	N	N	N	N	N	N	Y
	O	N	N	N	N	N	N	N	N	Y	N	N	Y
ALL	I	N	N	N	Y	Y	Y	N	N	N	Y	Y	Y
	IO	N	N	N	N	N	N	N	N	N	Y	Y	Y
	O	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y

```
Y       :2nd,3rd,...open is allowed
N       :2nd,3rd,...open is denied
DRW     :Deny Read/Write Mode (Exclusive)
DW      :Deny Write Mode
DR      :Deny Read Mode
RW      :Read/Write Mode
I       :Read Only Access
O       :Write Only Access
I/O     :Read/Write Access
```

3EH
Close a File Handle

Purpose:

Closes the specified file handle.

On Entry	Register Contents
AH	3EH
BX	File handle returned by open or create

On Return	Register Contents
AX	Error codes if carry flag set NONE if carry flag not set

Remarks:

On entry, BX contains the file handle that was returned by "open" or "create." On return, the file is closed, the directory is updated, and all internal buffers for that file are flushed.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

3FH
Read from a File or Device

Purpose:

Transfers the specified number of bytes from a file into a buffer location.

On Entry	Register Contents
AH	3FH
BX	File handle
DS:DX	Buffer address
CX	Number of bytes to be read

On Return	Register Contents
AX	Number of bytes read Error codes if carry flag set

Remarks:

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.

3FH
Read from a File or Device

This function call attempts to transfer (CX) bytes from a file into a buffer location. It is not guaranteed that all bytes will be read. For example, when DOS reads from the keyboard, at most one line of text is transferred. If this read is performed from the standard input device, the input can be redirected. See "Redirection of Standard Input and Output" in the *DOS Reference*. If the value in AX is 0, then the program has tried to read from the end of file.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: Requires Read access rights.

40H
Write to a File or Device

Purpose:

Transfers the specified number of bytes from a buffer into a specified file.

On Entry	Register Contents
AH	40H
BX	File handle
DS:DX	Address of the data to write
CX	Number of bytes to write

On Return	Register Contents
AX	Number of bytes written Error codes if carry flag set

Remarks:

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.

This function call attempts to transfer (CX) bytes from a buffer into a file. AX returns the number of bytes actually written. If the carry flag is not set and this value is not the same as the number requested (in CX), 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

40H
Write to a File or Device

"Redirection of Standard Input and Output" in the *DOS Reference.*

To truncate a file at the current position of the file pointer, set the number of bytes (CX) to zero before issuing the interrupt 21H. The file pointer can be moved to the desired position by reading, writing, and performing function call 42H (Move File Read/Write Pointer).

If the file is read – only, the write to the file or device is not performed.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: Requires Write access rights.

41H
Delete a File from a Specified Directory (UNLINK)

Purpose:

Removes a directory entry associated with a filename.

On Entry	Register Contents
AH	41H
DS:DX	Address of an ASCIIZ string

On Return	Register Contents
AX	Error codes if carry flag set NONE if carry flag not set

Remarks:

Global filename characters are not allowed in any part of the ASCIIZ string. Read – only files cannot be deleted by this call. To delete a read – only file, you can first use call 43H to change the file's read – only attribute to 0, then delete the file.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

41H
Delete a File from a Specified Directory (UNLINK)

Network Access Rights: Requires Create access rights.

42H
Move File Read Write Pointer (LSEEK)

Purpose:

Moves the read/write pointer according to the method specified.

On Entry	Register Contents
AH	42H
CX:DX	Distance (offset) to move in bytes
AL	Method of moving (0, 1, 2)
BX	File handle

On Return	Register Contents
AX	Error codes if carry flag set
DX:AX	New pointer location if carry flag not set

Remarks:

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).

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on

42H
Move File Read Write Pointer (LSEEK)

page 6-42 for more information on the codes returned from function call 59H.

This function call moves the read/write pointer according to the following methods:

AL	Description
0	The pointer is moved CX:DX bytes (offset) from the beginning of the file.
1	The pointer is moved to the current location plus offset.
2	The pointer is moved to the end-of-file plus offset. This method can be used to determine file's size.

Note: If an LSEEK operation is performed on a file that resides on a network disk that is open in either Deny Read or Deny None sharing mode, the read/write pointer information is adjusted on the computer where the file actually exists. If the file is opened in any other sharing mode, the read/write pointer information is kept on the remote computer.

43H
Change File Mode (CHMOD)

Purpose:

Changes the file mode of the specified mode.

On Entry	Register Contents
AH	43H
DS:DX	Pointer to an ASCIIZ path name
CX	Attribute
AL	Function code

On Return	Register Contents
AX	Error codes if carry flag set
CX	The file's current attribute; if carry flag not set and getting the attribute

Remarks:

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 01H, the file's attribute is set to the attribute in CX. See "DOS Disk Directory" on page 5-10 for the attribute byte description. If AL is 00H then the file's current attribute is returned in CX.

43H
Change File Mode (CHMOD)

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

> **Note:** The volume label and subdirectory bits of an attribute cannot be changed using CHMOD. When setting a file's attribute, bits 3 and 4 of CX must be zero, otherwise an error is indicated.

Network Access Rights: To change the archive bit (AL = 20H), no access rights are required. To change any other bit, Create access rights are required.

44H
I/O Control for Devices (IOCTL)

Purpose:

Sets or gets device information associated with open device handles, or sends control strings to the device handle or receives control strings from the device handle.

Remarks:

The following function values are allowed in AL:

AL = 00H Get device information (returned in DX).

AL = 01H Set device information (determined by DX). Currently, DH must be zero for this call.

AL = 02H Read from character device

AL = 03H Write to character device

AL = 04H Read from block device

AL = 05H Write to block device

AL = 06H Get input status.

AL = 07H Get output status.

AL = 08H Is a particular block device changeable?

AL = 09H Is a logical device local or remote?

AL = 0AH Is a handle local or remote?

AL = 0BH Change sharing retry count

44H
I/O Control for Devices (IOCTL)

AL = 0CH Generic IOCTL handle request (code page switching)

AL = 0DH Block device Generic IOCTL request

AL = 0EH Get logical drive

AL = 0FH Set logical drive

IOCTL can be used to get information about device channels. You can make calls on regular files, but only function values 00H, 06H, and 07H are defined in that case. All other calls return an "Invalid Function" error.

Function values 00H to 08H are not supported on network devices. Function value 0BH requires the file sharing command to be loaded (SHARE).

Calls AL = 00H and AL = 01H

Purpose:

Sets or gets device information.

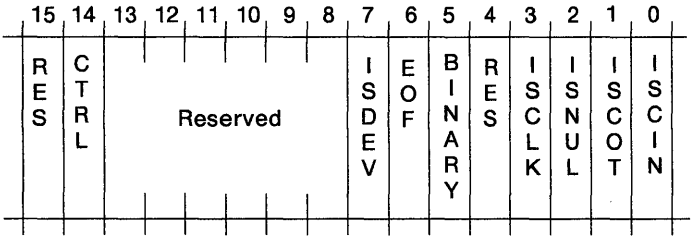
On Entry	Register Contents
AH	44H
AL	00H - Get device information 01H - Set device information
BX	Handle
DH	0 for AL = 01H
DL	Device information

44H
I/O Control for Devices (IOCTL)

On Return	Register Contents
DX	Device information

Remarks:

The bits of DX are defined as follows:



ISDEV = 1 if this channel is a device.

= 0 if this channel is a disk file (bits 8-15 are 0 in this case).

Bits 8-15 of DX correspond to the upper 8 bits of the device driver Attribute word. See chapter 2 for details.

If ISDEV = 1

EOF = 0 if end-of-file on input.

BINARY = 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=02H, AL=03H, AL=04H, and AL=05H.

CTRL = 1 if this device can process control strings via calls AL=02H and AL=03H. Note that this bit cannot be set by function call 44H.

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=01H.

44H
I/O Control for Devices (IOCTL)

Calls AL = 02H, AL = 03H

Purpose:

These two calls allow arbitrary control strings to be sent or received from a character device.

On Entry	Register Contents
AH	44H
AL	02H - Read from character device 03H - Write to character device
DS:DX	Data or buffer
CX	Number of bytes to read or write
BX	Handle

On Return	Register Contents
AX	Number of bytes transferrred

Remarks:

These are the Read and Write calls for a character device. An "Invalid Function" error is returned if the CTRL bit is zero.

44H
I/O Control for Devices (IOCTL)

Calls AL = 04H, AL = 05H

Purpose:

These two calls allow arbitrary control strings to be sent or received from a block device.

On Entry	Register Contents
AH	44H
AL	04H - Read from block device 05H - Write to block device
DS:DX	Data or buffer
CX	Number of bytes to read or write
BL	Drive number (0 = default, 1 = A, etc.)

On Return	Register Contents
AX	Number of bytes transferrred

Remarks:

These are the Read and Write calls for a block device. The drive number is in BL for these calls An "Invalid Function" error is returned if the CTRL bit is zero. An "Access-Denied" code is returned if the drive is invalid.

I/O Control for Devices (IOCTL)

Calls AL = 06H and AL = 07H

Purpose:

These calls allow you to check if a handle is ready for input or output.

On Entry	Register Contents
AH	44H
AL	06H - Get input status 07H - Get output status
BX	Handle

On Return	Register Contents
AL	For a file: FFH until end-of-file is reached 00H after end-of-file is reached For devices: 00H - not ready 0FH - ready

Remarks:

If used for a file, AL always returns FFH until end-of-file is reached, then always returns 00H unless the current file position is changed through call 42H. When used for a device, AL returns FFH for ready or zero for not ready.

44H
I/O Control for Devices (IOCTL)

Call AL = 08H (DOS 3.00 to 3.30)

Purpose:

This call allows you to determine if a device can support removable media.

On Entry	Register Contents
AH	44H
AL	08H - Is device removable?
BL	Drive number (0 = default, 1 = A, 2 = B, 3 = C, etc.)

On Return	Register Contents
AX	00H if device is removable 01H if device is fixed 0FH if the value in BL is invalid

Remarks:

If the value returned in AX is 0, then the device is removable. If the value is 1, then the device is fixed. The drive number should be placed in BL. If the value in BL is invalid, then error code 0FH is returned. For network devices, the error **Invalid function** is returned.

44H
I/O Control for Devices (IOCTL)

Call AL = 09H (DOS 3.10 to 3.30)

Purpose:

This call allows you to determine if a logical device is associated with a network directory.

On Entry	Register Contents
AH	44H
AL	09H - Is device local or remote?
BL	Drive number (0 = default, 1 = A, etc.)

On Return	Register Contents
DX	For local devices, bit 12 is 0. For remote devices, bit 12 is set.

Remarks:

On entry,

BL contains the drive number of the block device you want to check (0 = default, 1 = A, 2 = B, and so forth). The value returned in DX indicates whether the device is local or remote. Bit 12 is set for remote devices (1000H). Bit 12 is not set for local devices. The other bits in DX are reserved. If disk redirection is paused, the function returns with bit 12 not set.

44H
I/O Control for Devices (IOCTL)

Call AL = 0AH (3.10 to 3.30)

Purpose:

This call allows you to determine if a handle is for a local device or a remote device across the network.

On Entry	Register Contents
AH	44H
AL	0AH - Is handle local or remote?
BX	File handle

On Return	Register Contents
DX	For local devices, bit 15 is 0. For remote devices, bit 15 is set.

Remarks:

For remote devices, bit 15 is set (8000H). The handle should be placed in BX. Bit 15 is not set for local devices.

44H
I/O Control for Devices (IOCTL)

Call AL = 0BH (DOS 3.00 to 3.30)

Purpose:

Controls retries on sharing and lock resource conflicts.

On Entry	Register Contents
AH	44H
AL	0BH - Change sharing retry count
CX	Number of times to execute a delay loop
DX	Number of retries

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

All sharing and lock conflicts are automatically retried a number of times before they are returned as a DOS error or critical error. You can select the number of retries and the delay time between retries. On input, CX contains the number of times to execute a delay loop, and DX contains the number of retries. The delay loop consists of the following sequence:

```
XOR     CX,CX
LOOP    $           ;spin 64K times
```

If this call is never issued, DOS uses delay = 1 and retries = 3 as the defaults for CX and DX. If you expect your application to cause sharing or lock

conficts on locks that are in effect for a short period of time, you may want to increase the values for CX and DX to minimize the number of errors actually returned to your application.

Call AL = 0CH (DOS 3.30.)

This call allows a device driver to support a new set of subfunction calls that implement code page switching. The format for these calls is:

44H
I/O Control for Devices (IOCTL)

On Entry	Contents
AH	44H - IOCTL Request
AL	0CH - Handle Generic IOCTL (Code page switching)
BX	Handle - (handle of open device)
CH	Major codes 00 - Unknown 01 - COMx 03 - CON 05 - LPTx
CL	Function within category code (Minor code) 4CH = Prepare start 4DH = Prepare end 4AH = Select 6AH = Query selected 6BH = Query prepare list
DS:DX	Pointer to parameter block

Refresh requests the device driver to set up the device with the most recently prepared codepage. The refresh operation is requested by doing a prepare start with all codepage values (prep_strt_pkcp?) set to a -1. This operation must be followed immediately by a prepare end.

Remarks:

When CL = 4AH, 4DH or 6AH the parameter block, pointed to by DS:DX, has the following layout:

```
packet      struc
packlen     dw   2   ;Length of packet in bytes
packcpid    dw   ?   ;Code page ID
packet      ends
```

Remarks:

When CL = 4CH, the parameter block, pointed to by DS:DX, has the following layout:

```
prep_strt_packet   struc
prep_strt_pkfl     dw   0    ;flags
prep_strt_pklen    dw   (n+1)*2  ;byte length
                                 ;of rest
                                 ;of packet,not
                                 ;including this
                                 ;length field
prep_strt_pknum    dw   n    ;number of code pages
                             ;in the following list
prep_strt_pkcp1    dw  -1    ;code page one
prep_strt_pkcp2    dw  -1    ;code page two
                     .
                     .
                     .
                     .
prep_strt_pkcp?    dw  -1    ;code page n
prep_strt_packet   ends
```

Notes:

1. A -1 for any prep_strt_pkcp?, tells the device driver not to change the code page value for that position. Any other value is a codepage to be prepared.

2. n is the number of additional code pages specified in the DEVICE= COMMAND in CONFIG.SYS. See chapter 4 in the *DOS 3.30 Reference* for more information. The value for n can be up to 12.

3. For cartridge-prepares set the **prep_strt_pkfl** field to 1.

44H
I/O Control for Devices (IOCTL)

Remarks:

When CL = 6AH, the parameter block, pointed to by DS:DX, has the following layout:

```
query_list_packet        struc
query_list_packet_len    dw     ((n+1)+(m+1))*2
hrdw_codpage_count       dw     n
hrdw_codepage_1          dw     -1
                         .
                         .
                         .
                         .
hrdw_codepage_n          dw     -1
prepd_codepage_count     dw     m
prepd_codepage_1         dw     -1
                         .
                         .
                         .
                         .
prepd_codepage_m         dw     -1
query_list_packet        ends
```

Note: The device driver may return up to 12 code page values for each type of code page (hardware or prepared) so "n" can be up to 12, and "m" can be up to 12.

On Return	Register Contents
Carry flag	Set if error occurs. (Issue a "Get Extended Error" to get error code.)

Many of the function calls return the carry flag clear if the operation was successful. If an error condition was encountered, the carry flag is set. To obtain information about the error, such as the *error class*, *locus*, and recommended *action*, issue the Get Extended Error function call 59H.

The list that follows contains the possible error codes returned from Get Extended Error for code page switching operations.

PREPARE Start Error Codes:

Code	Meaning
01	Invalid function number
27	Code page conflict (used for keyb xx mismatch)
29	Device error
22	Unknown command

PREPARE Write Error Codes:

Code	Meaning
27	Device not found in file, or code page not found in file
29	Device error
31	File contents not a font file, or file contents structure damaged

PREPARE End Error Codes:

Code	Meaning
19	Bad data read from font file
31	No prepare start

44H
I/O Control for Devices (IOCTL)

SELECT Error Codes:

Code	Meaning
26	Code page not prepared
27	Current keyb does not support this code page
29	Device error

QUERY Selected Error Codes:

Code	Meaning
26	No code page has been selected
27	Device error

QUERY Prepared List Error Codes:

Code	Meaning
26	No code pages have been selected
29	Device error

REFRESH Error Codes Returned From PREPARE Start For a REFRESH Request:

Code	Meaning
27	Keyboard/code page conflict
29	Device error
31	Device driver does not have copy of code page to download to device

44H
I/O Control for Devices (IOCTL)

After a Prepare start, data defining the codepage font is written to the driver using one or more IOCTL write control string (AX=4403H) calls. This is assumed to be information to download to the device. The stream is ended by a Prepare end. The stream format is device specific.

If no data is written for a prepare operation, the driver is to interpret the newly prepared code page(s) as a hardware code page. This allows devices that support user changeable hardware fonts (usually in cartridges) to be supported.

Note: No prepare is needed for hardware-defined code pages.

Call AL = 0DH (DOS 3.20 and 3.30)

Purpose:

The Generic IOCTL request tells block device drivers to perform one of the following functions:

- Get Device Parameters
- Set Device Parameters
- Read Track on a Logical Device
- Write Track on a Logical Device
- Format and Verify Track on a Logical Device
- Verify Track on a Logical Device

44H
I/O Control for Devices (IOCTL)

Remarks:

CH contains the Major code (08H for all functions) and CL contains the minor code (function.)

44H
I/O Control for Devices (IOCTL)

The following register descriptions are valid for each of the functions described on the previous page.

On Entry	Contents
AH	44H - IOCTL Request
AL	0DH - Generic IOCTL Request
BL	Drive Number (0 = default, 1 = A, etc.)
CH	08H - category code (Major code)
CL	Function within Category code (Minor code) 40H = Set device parameters 60H = Get device parameters 41H = Write track on logical device 61H = Read track on logical device 42H = Format and verify track on a logical device 62H = Verify track on a logical device
DS:DX	Pointer to parameter block

44H
I/O Control for Devices (IOCTL)

Get or Set Device Parameters

To Get device parameters, set CL = 60H.

To Set device parameters, set CL = 40H.

Remarks:

When CL = 60H or CL = 40H, the parameter block has the following field layout:

```
A_deviceParameters      struc
SpecialFunctions        db      ?
DeviceType              db      ?
DeviceAttributes        dw      ?
NumberOfCylinders       dw      ?
MediaType               db      ?
DeviceBPB               a_BPB   <>
TrackLayout             a_TrackLayout <>
A_deviceParameters      ends
```

An explanation of each field in the parameter block is given in the pages that follow.

44H
I/O Control for Devices (IOCTL)

SpecialFunctions Field:

This 1-byte field is used to further define the Get and Set Device Parameters functions.

For the Get Device Parameters function, bit 0 of the **SpecialFunctions** field has the following meaning:

Bit 0 =1 Return the BPB that BUILD BPB would return.
=0 Return the default BPB for the device.

Note: All other bits must be off.

For the Set Device Parameters function bits 0, 1, and 2 of the **SpecialFunctions** field are used.

These bits have the following meanings when CL = 40H.

Bit 0 =1 All subsequent BUILD BPB requests return **DeviceBPB**. If another Set Device request is received with bit 0 reset, BUILD BPB returns the actual media BPB.

=0 Indicates that the **DeviceBPB** field contains the new default BPB for this device. If a previous Set Device request set this bit on, the actual media BPB is returned. Otherwise, the default BPB for the device is returned by BUILD BPB.

SpecialFunctions field continued:

Bit 1 =1 Ignore all fields in the Parameter Block except the **TrackLayout** field.

=0 Read all fields of the parameter block.

Bit 2 =1 Indicates that all sectors in the track are the same size and all sector numbers are between 1 and *n* (where *n* is the number of sectors in the track.)

=0 Indicates that all sectors in the track may not be the same size.

Notes:

1. All other bits must be reset.

2. Set bit 2 for normal track layouts. Format Track can be more efficient if bit 2 is set.

3. Setting bits 0 and 1 at the same time is invalid and should be considered an error.

DeviceType Field:

A 1-byte field that describes the physical device type. Device type is not set by IOCTL but is received from the device.

The values in this field have the following meanings:

0 = 320/360 KB 5-1/4-inch
1 = 5-1/4-inch, 1.2 MB
2 = 3-1/2-inch, 720 KB
3 = 8 inch single density
4 = 8 inch double density
5 = Fixed disk
6 = Tape drive
7 = Other

DeviceAttributes Field:

A 1-byte field that describes the physical attributes of the device. Device attributes are not set by IOCTL but are received from the device driver.

Only bits 0 and 1 of this field are used. They have the following meanings:

Bit 0 = 1 media is not removable.
= 0 media is removable.
Bit 1 = 1 diskette changeline is supported.
= 0 diskette changeline is not supported.

Bits 2 -7 are reserved.

NumberOfCylinders Field:

This field indicates the maximum number of cylinders that can be supported on the physical device, independent of the media type. The information in this field is not set by IOCTL, but is received from the device driver.

MediaType Field:

For multimedia drives, this field indicates which media is expected to be in the drive.

The **MediaType** field is used only when the actual media in the drive cannot otherwise be determined. Media type is dependent on device type.

Regardless of the device type, a value of 0 represents the default. For example, a 5-1/4-inch 1.2MB diskette drive is a multi-media drive. The media type is defined as follows:

0 = Quad density 1.2 MB (96 tpi) diskette

1 = Double density 320/360KB (48 tpi) diskette

The default media type for a 1.2MB drive is a quad density 1.2 MB diskette.

DeviceBPB Field:

For the Get Device Parameters function:

- If bit 0 of the **SpecialFunctions** field is set, the device driver returns the BPB that BUILD BPB would return.
- If bit 0 of the **SpecialFunctions** field is not set, the device driver returns the default BPB for the device.

For the Set Device Parameters function:

- If bit 0 of the **SpecialFunctions** field is set, the device driver is requested to return the BPB from this field for all subsequent BUILD BPB requests until a Set Device Parameters request is received with bit 0 in the **SpecialFunctions** field reset.
- If bit 0 is not set, the BPB contained in this field becomes the new default BPB for the device.

The **DeviceBPB** field has the following format:

```
a_BPB                struc
BytesPerSector       dw      ?
SectorsPerCluster    db      ?
ReservedSectors      dw      ?
NumberOfFATs         db      ?
RootEntries          dw      ?
TotalSectors         dw      ?
MediaDescriptor      db      ?
SectorsPerFAT        dw      ?
;
SectorsPerTrack      dw      ?
Heads                dw      ?
HiddenSectors        dd      ?
Reserved_1           dd      ?
Reserved_2           db      6 dup (0)
a_BPB                ends
```

TrackLayout Field:

This is a variable length table indicating the expected layout of sectors on the media track.

DOS device drivers do not keep a track layout table for each logical device. The global track table must be updated (via Set Device Parameters) when the attributes of the media change.

> **Note:** The Set Device Parameters call (CL = 40H) modifies the track table regardless of how bit 1 of the **SpecialFunctions** field is set.

For Get Device Parameters, this field is not used. The track layout is used by subsequent Read/Write Track, Format/Verify Track and Verify Track functions.

The following example shows how this field is formatted:

```
Total sectors-------|SectorCount      dw  n

Sector  1-----------|SectorNumber_1    dw  1H
                    |SectorSize_1      dw  200H

Sector  2-----------|SectorNumber_2    dw  2H
                    |SectorSize_2      dw  200H

Sector  3-----------|SectorNumber_3    dw  3H
                    |SectorSize_3      dw  200H

Sector  4-----------|SectorNumber_4    dw  4H
                    |SectorSize_4      dw  200H
                                |
                                |
                                |
Sector  n-----------|SectorNumber_n    dw  n
                    |SectorSize_n      dw  200H
```

Note: All values are in hexadecimal.

The total number of sectors is indicated by the **SectorCount** field. Each sector number must be unique and in a range between 1 and n (sector count). As shown, in the example above, the first sector number is 1 and the last sector number is equal to the sector count (n). If bit 2 of the **SpecialFunctions** field is set, all sector sizes, which are measured in bytes, must be the same, (See the example).

See the description of bit 2 under the **SpecialFunction** field.

Note: The **DeviceType**, **DeviceAttributes**, and **NumberOfCylinders** fields should be changed only if the physical device has been changed.

44H
I/O Control for Devices (IOCTL)

Read/Write Track on Logical Device

To read a track on a logical device, set CL = 61.

To write a track on a logical device, set CL = 41.

The parameter block has the following layout when reading or writing a track on a logical device.

```
a_ReadWriteTrackPacket  struc
SpecialFunctions        db      ?
Head                    dw      ?
Cylinder                dw      ?
FirstSector             dw      ?
NumberOfSectors         dw      ?
TransferAddress         dd      ?
A_ReadWriteTrackPacket  ends
```

Notes:

1. All bits in the **SpecialFunctions** field must be reset.

2. The value in the **FirstSector** field and the **NumberOfCylinders** field is zero-based. For example, to indicate sector 9, set the value to 8.

I/O Control for Devices (IOCTL)

Format/Verify Track on Logical Drive (IOCTL Write)

To format and verify a track, set CL = 42H.

To verify a track, set CL = 62H.

The parameter block has the following layout when formatting a track or verifying a track on a logical drive.

```
A_FormatPacket          struc
SpecialFunctions        db      ?
Head                    dw      ?
Cylinder                dw      ?
A_FormatPacket          ends
```

On entry, bit 0 of the SpecialFunctions field has the following meanings:

Bit 0 = 1 Format status check call to determine if a combination of number-of-tracks and sectors-per-track is supported.

= 0 Format /Verify track call.

To determine if a combination of number-of-tracks and sectors- per-track is supported, a Set Device Parameters call must be issued with the correct BPB for that combination before issuing the Format Status call. The device driver can then return the correct code to indicate what is supported.

The values returned in the SpecialFunctions field for a Format Status Check call are:

0 = This function is supported by the ROM BIOS. The specified combination of number-of-tracks and sectors-per-track is

allowed for the diskette drive.

1 = This function is not supported by the ROM BIOS.

2 = This function is supported by the ROM BIOS. The specified combination of number-of-tracks and sectors-per-track is not allowed for the diskette drive.

3 = This function is supported by the ROM BIOS, but ROM BIOS cannot determine if the numbers-of-tracks and sectors-per-track are allowed because the diskette drive is empty.

To format a track:

1. Issue the Set Device Parameters function call.

2. Issue the Format Status Check function call to validate the number-of-tracks and sectors-per-track combination. Ignore the result if the value returned is 1, because the ROM BIOS does not support this function.

3. Issue the Format/Verify Track function call with the SpecialFunctions bit 0 reset for each track on the medium.

44H
I/O Control for Devices (IOCTL)

Call AL = 0EH (DOS 3.20 and 3.30)

Purpose:

This call allows the device driver to determine if more than one logical drive is assigned to a block device. When this call is issued, a drive number is passed in BL on input.

On Entry	Register Contents
AH	44H
AL	0EH
BL	Drive number (0 = default, 1 = A, etc.)

On Return	Register Contents
AL	(0 = only one letter assigned to this block device, otherwise 1 = A, 2 = B, 3 = C, etc.)
AX	Error code if carry flag set

Remarks:

If the block device has more than one logical drive letter assigned to it, on output a drive number corresponding to the last drive letter that was used to reference the device is returned in AL. If only one drive letter is assigned to the device, 0 is returned in AL by this call.

44H
I/O Control for Devices (IOCTL)

Call AL = 0FH (DOS 3.20 and 3.30)

Purpose:

This call requests the device driver to set the next logical drive letter that will be used to reference a block device.

On Entry	Register Contents
AH	44H
AL	0FH
BL	Drive number

On Exit	Register Contents
AL	0 = Only one drive letter assigned to the block device (otherwise, 1 = A, 2 = B, etc.)
AX	Error code if carry set

Remarks:

When copying diskettes on a drive whose physical drive number has more than one logical drive letter assigned to it (for example, copying on a single drive system) DOS issues diskette swap prompts to tell you which logical drive letter is currently referencing the physical drive number. As the drive changes from source to target, DOS issues the message:

```
"Insert diskette for drive X: and strike any key when ready."
```

It is possible to avoid this message by issuing call AL = 0FH (Set Logical Drive).

To avoid the DOS diskette swap message, set BL to the drive number that corresponds to the drive letter that will be referenced in the next I/O request.

> **Note:** You can determine the last logical drive letter assigned to the physical drive number by issuing call AL = 0EH.

Because any block device can have logical drives, this call should be issued before all I/O operations involving more than one drive letter; otherwise, the DOS message may be issued.

45H
Duplicate a File Handle (DUP)

Purpose:

Returns a new file handle for an open file that refers to the same file at the same position.

On Entry	Register Contents
AH	45H
BX	File handle

On Return	Register Contents
AX	New file handle if carry flag not set Error codes if carry flag set

Remarks:

On entry, BX contains the file handle. On return, AX contains the returned file handle.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

> **Note:** If you move the read/write pointer of either handle by a read, write, or LSEEK function call, the pointer for the other handle is also changed.

46H
Force a Duplicate of a Handle (FORCDUP)

Purpose:

Forces the handle in CX to refer to the same file at the same position as the handle in BX.

On Entry	Register Contents
AH	46H
BX	Existing file handle
CX	Second file handle

On Return	Register Contents
AX	Error codes if carry flag set None if carry flag not set

Remarks:

On entry, BX contains the file handle. CX contains a second file handle. On return, the CX file handle refers to the same file at the same position as the BX file handle. If the CX file handle was an open file, then it is closed first. If you move the read/write pointer of either handle, the pointer for the other handle is also changed.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on

46H
Force a Duplicate of a Handle (FORCDUP)

page 6-42 for more information on the codes returned from function call 59H.

47H
Get Current Directory

Purpose:

Places the full path name (starting from the root directory) of the current directory for the specified drive in the area pointed to by DS:SI.

On Entry	Register Contents
AH	47H
DS:SI	Pointer to a 64-byte user memory area
DL	Drive number (0 = default, 1 = A, etc.)

On Return	Register Contents
DS:SI	Filled out with full path name from the root if carry is not set
AX	Error codes if carry flag is set

Remarks:

The drive letter is not part of the returned string. The string does not begin with a backslash and is terminated by a byte containing 00H.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on

page 6-42 for more information on the codes returned from function call 59H.

48H
Allocate Memory

Purpose:

Allocates the requested number of paragraphs of memory.

On Entry	Register Contents
AH	48H
BX	Number of paragraphs of memory requested

On Return	Register Contents
AX:0	Points to the allocated memory block
AX	Error codes if carry set
BX	Size of the largest block of memory available (in paragraphs) if the allocation fails

Remarks:

On entry, BX contains the number of paragraphs requested. On return, AX:0 points to the allocated memory block. If the allocation fails, BX returns the size of the largest block of memory available in paragraphs.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on

page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

49H
Free Allocated Memory

Purpose:

Frees the specified allocated memory.

On Entry	Register Contents
AH	49H
ES	Segment of the block to be returned

On Return	Register Contents
AX	Error codes if carry flag set NONE if carry flag not set

Remarks:

On entry, ES contains the segment of the block to be returned to the system pool. On return, the block of memory is returned to the system pool.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

4AH
Modify Allocated Memory Blocks (SETBLOCK)

Purpose:

Modifies allocated memory blocks to contain the new specified block size.

On Entry	Register Contents
AH	4AH
ES	Segment of the block
BX	Contains the new requested block size in paragraphs

On Return	Register Contents
AX	Error codes if carry flag set None if carry flag not set
BX	Maximum poolsize possible if the call fails on a "grow request" if carry flag is set

Remarks:

DOS attempts to "grow" or "shrink" the specified block.

Error codes are returned in AX. Issue function call 59H "Get Extended Error" for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on

4AH
Modify Allocated Memory Blocks (SETBLOCK)

page 6-42 for more information on the codes returned from function call 59H.

4BH
Load or Execute a Program (EXEC)

Purpose:

Allows a program to load another program into memory and optionally begins execution of it.

On Entry	Register Contents
AH	4BH
DS:DX	Points to the ASCIIZ string with the drive, path, and filename to be loaded
ES:BX	Points to a parameter block for the load
AL	Function value (see description)

On Return	Register Contents
AX	Error codes if carry flag set NONE if carry flag not set

Remarks:

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

4BH
Load or Execute a Program (EXEC)

The following function values are allowed in AL:

Function Value	Description
00H	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.
03H	Load, do not create the program segment prefix, and do not begin execution. This is useful in loading program overlays.

Load or Execute a Program (EXEC)

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

AL = 00H 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

Note: The DWORD pointers are in offset segment form.

AL = 03H Load overlay

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

4BH
Load or Execute a Program (EXEC)

All open files of a process are duplicated in the newly created process after an EXEC, except if the file was opened with the inheritance bit set to 1. This means that 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

Following the byte of zero in the environment, is a WORD that indicates the number of other strings following. Following this is a copy of the DS:DX filename passed to the child process. For example, the string VERIFY = ON could be passed. A zero value of the environment address causes the newly created process to inherit the parent's environment unchanged. The segment address of the environment

is placed at offset 2CH of the program segment prefix for the program being invoked.

Errors codes are returned in AX. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned.

Notes:

1. When your program received control, all available memory was allocated to it. You must free some memory (see call 4AH) 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.

4CH
Terminate a Process (EXIT)

Purpose:

Terminates the current process and transfers control to the invoking process.

On Entry	Register Contents
AH	4CH
AL	Return code

On Return	Register Contents
	NONE

Remarks:

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 4DH. All files opened by this process are closed.

4DH
Get Return Code of a Subprocess (WAIT)

Purpose:

Gets the return code specified by another process either through function call 4CH or function call 31H. It returns the Exit code only once.

On Entry	Register Contents
AH	4DH

On Return	Register Contents
AX	Return code

Remarks:

The low byte of the exit code contains the information sent by the exiting routine. The high byte of the exit code can contain:

- 00H – for normal termination
- 01H – for termination by Ctrl-Break
- 02H – for termination as a result of a critical device error
- 03H – for termination by call 31H

4EH
Find First Matching File (FIND FIRST)

Purpose:

Finds the first filename that matches the specified file specification.

On Entry	Register Contents
AH	4EH
DS:DX	Pointer to an ASCIIZ string containing the drive, path, and filename of the file to be found
CX	Attribute used in searching for the file

On Return	Register Contents
AX	Error codes if carry flag set

Remarks:

The filename in DS:DX can contain global filename characters. The ASCIIZ string cannot contain a network path. See function call 11H for a description of how the attribute bits are used for searches.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more

information on the codes returned from function call 59H.

If a file is found that matches the specified drive, path, and filename and attribute, the current DTA is filled in as follows:

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

1 byte - file's attribute

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.

4FH
Find Next Matching File (FIND NEXT)

Purpose:

Finds the next directory entry matching the name that was specified on the previous Find First or Find Next function call.

On Entry	Register Contents
AH	4FH
DTA	Contains the information from a previous Find First or Find Next call (4EH, 4FH)

On Return	Register Contents
AX	Error codes if carry flag set

Remarks:

If a matching file is found, the DTA is set as described in call 4EH. If no more matching files are found, an error code is returned.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

54H
Get Verify Setting

Purpose:

Returns the value of the verify flag.

On Entry	Register Contents
AH	54H

On Return	Register Contents
AL	Current verify flag value 00H, if verify is off 01H, if verify is on

Remarks:

On return, AL returns 00H if verify is OFF, 01H if verify is ON. Note that the verify switch can be set through call 2EH.

56H
Rename a File

Purpose:

Renames the specified file.

On Entry	Register Contents
AH	56H
DS:DX	Pointer to an ASCIIZ string containing the drive, path, and filename of the file to be renamed
ES:DI	Pointer to an ASCIIZ string containing the new path and filename

On Return	Register Contents
AX	Error codes if carry flag set NONE if carry flag not set

Remarks:

If a drive is used in the ASCIIZ 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. Global filename characters are not allowed in the filename.

56H
Rename a File

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: Requires Create access rights.

57H Get/Set a File's Date and Time

Purpose:

Gets or sets a file's date and time.

On Entry	Register Contents
AH	57H
AL	00H, get date and time 01H, set date and time
BX	File handle
CX	Time to be set if AL = 01H
DX	Date to be set if AL = 01H

On Return	Register Contents
AX	Error codes if carry flag set
DX	If getting date, the date is from the handle's internal table
CX	If getting time, the time is from the handle's internal table

Remarks:

The date and time formats are the same as those for the directory entry described in Chapter 5 of this book.

57H Get/Set a File's Date and Time

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

59H (DOS 3.00 to 3.30)
Get Extended Error

Purpose:

Returns additional error information, such as the error class, locus, and recommended action.

On Entry	Register Contents
AH	59H
BX	0000H (version 0, for 3.00 to 3.30)

On Return	Register Contents
AX	Extended error
BH	Error class
BL	Suggested action
CH	Locus

Remarks:

This function call returns the error class, locus, and recommended action, in addition to the return code. Use this function call from:

- Interrupt 24H error handlers
- Interrupt 21H function calls that return an error in the carry bit
- FCB function calls that return FFH

59H (DOS 3.00 to 3.30)
Get Extended Error

On return, the registers contents of DX, SI, DI, ES, CL, and DS are destroyed.

59H (DOS 3.00 to 3.30)
Get Extended Error

Error Return in Carry Bit

For function calls that indicate an error by setting the carry flag, the correct method for performing function call 59H is:

1. Load up registers.
2. Issue interrupt 21H.
3. Continue operation, if carry not set.
4. Disregard the error code and issue function call 59H to obtain additional information.
5. Use the value in BL to determine the suggested action to take.

Error Status in AL

For function calls that indicate an error by setting AL to FFH, the correct method for performing function call 59H is:

1. Load up registers.
2. Issue interrupt 21H.
3. Continue operation, if error is not reported in AL.
4. Disregard the error code and issue function call 59H to obtain additional information.
5. Use the action in BL to determine the suggested action to take.

5AH (DOS 3.00 to 3.30)
Create Unique File

Purpose:

Generates a unique filename, and creates that file in the specified directory.

On Entry	Register Contents
AH	5AH
DS:DX	Pointer to ASCIIZ path ending with a backslash (\)
CX	Attribute

On Return	Register Contents
AX	Error codes if carry flag is set
DS:DX	ASCIIZ path with the filename of the new file appended

Remarks:

On entry, AH contains 5AH. If no error has occurred, then the file is opened in compatibility mode with Read/Write access, the read/write pointer is set at the first byte of the file and AX contains the file handle and the filename is appended to the path specified in DS:DX.

This function call generates a unique name and attempts to create a new file in the specified directory. If the file already exists in the directory, then another unique name is generated and the process is repeated. Programs that need temporary

files should use this function call to generate unique filenames.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

> **Note:** The file created using this function call is not automatically deleted at program termination.

Network Access Rights: Requires Create access rights.

5BH (DOS 3.00 to 3.30)
Create New File

Purpose:

Creates a new file.

On Entry	Register Contents
AH	5BH
DS:DX	Pointer an ASCIIZ path name
CX	File attributes

On Return	Register Contents
AX	Error codes if carry flag set Handle if carry flag not set

Remarks:

This function call is identical to function call 3CH (Create) with the exception that it will fail if the filename already exists. The file is created in compatibility mode for reading and writing and the read/write pointer is set at the first byte of the file.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Network Access Rights: Requires Create access rights.

5CH (DOS 3.00 to 3.30)
Lock/Unlock File Access

Purpose:

Locks or unlocks a range of bytes in an opened file.

On Entry	Register Contents
AH	5CH
AL	00H, to lock 01H, to unlock
BX	File handle
CX	Offset high
DX	Offset low
SI	Length high
DI	Length low

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

The Lock/Unlock function calls should only be used when a file is opened using the Deny Read or Deny None sharing modes. These modes do no local buffering of data when accessing files on a network disk.

5CH (DOS 3.00 to 3.30)
Lock/Unlock File Access

AL = 00H Lock

Provides a simple mechanism for excluding other processes read/write access to regions of the file. If another process attempts to read or write in such a region, its system call is retried the number of times specified with the system retry count set by IOCTL. If after those retries no success occurs, a general failure error is generated signaling the condition. The number of retries, as well as the length of time between retries, can be changed using function call 440BH (IOCTL Change Sharing Retry Count). The recommended action is to issue function call 59H to get the error code in addition to the error class, locus, and recommended action. The locked regions can be anywhere in the logical file. Locking beyond end-of-file is not an error. It is expected that the time in which regions are locked will be short. Duplicating the handle duplicates access to the locked regions. Access to the locked regions is not duplicated across the EXEC system call. Exiting with a file open and having issued locks on that file has undefined results. Programs that may be aborted using INT 23H or INT 24H should trap these and release the locks before exiting. The proper method for using locks is not to rely on being denied read or write access, but attempting to lock the region desired and examining the error code.

AL = 01H Unlock

Unlock releases the lock issued in the lock system call. The region specified must be exactly the same as the region specified in the previous lock. Closing a file with locks still in force has undefined results. Exiting with a file open and having issued locks on that file has undefined results. Programs that may be aborted using INT 23H or INT 24H should trap these and release the lock before exiting. The proper method for using locks is not to rely on being denied

read or write access, but attempting to lock the region desired and examining the error code.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

5E00H (DOS 3.10 to 3.30)
Get Machine Name

Purpose:

Returns the character identifier of the local computer.

On Entry	Register Contents
AX	5E00H
DS:DX	Pointer to the memory buffer where the ASCIIZ computer name is returned

On Return	Register Contents
DS:DX	Filled with the ASCIIZ computer name
CH	Name/number indicator flag 0 = name not defined not 0 = name/number defined
CL	NETBIOS name number for the name
AX	Error codes if carry flag is set

Remarks:

Get Machine Name returns the text of the current computer name to the caller. The computer name is a 15-character byte string padded with spaces and followed by a 00H byte. If the computer name was never set, register CH is returned with 00H and the value in the CL register is invalid. The IBM PC

5E00H (DOS 3.10 to 3.30)
Get Machine Name

Local Area Network Program must be loaded for the function call to execute properly.

5E02H (DOS 3.10 to 3.30)
Set Printer Setup

Purpose:

Specifies an initial string for printer files.

On Entry	Register Contents
AX	5E02H
BX	Redirection list index
CX	Length of setup string (maximum length is 64 bytes)
DS:SI	Pointer to printer setup buffer

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

The string specified is put in front of all files destined for a particular network printer. Printer Setup allows multiple users of a single printer to specify their own mode of operation for the printer. BX is set to the same index that is used in function call 5F02H (Get Redirection List Entry). An error code is returned if print redirection is paused or if the IBM PC Local Area Network Program is not loaded.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more

5E02H (DOS 3.10 to 3.30)
Set Printer Setup

information on the codes returned from function call 59H.

IMPORTANT: The redirection index value may change if function call 5F03H (Redirect Device) or function call 5F04H (Cancel Redirection) is issued between the time the redirection list is scanned and function call 5E02H (Set printer setup) is issued. Therefore, we recommend that you issue Set Printer Setup immediately after you issue "Get Redirection List."

5E03H (DOS 3.10 to 3.30)
Get Printer Setup

Purpose:

Returns the printer setup string for printer files.

On Entry	Register Contents
AX	5E03H
BX	Redirection list index
ES:DI	Pointer to printer setup buffer (maximum length is 64 bytes)

On Return	Register Contents
AX	Error codes if carry flag is set
CX	Length of data returned
ES:DI	Filled with the printer setup string

Remarks:

This function call returns the printer setup string which was specified using the function call 5E02H (Set Printer Setup). The setup string is attached to all files destined for a particular printer. The value in BX is set to the same Entry). Error code 1 (invalid function number) is returned if the IBM PC Local Area Network is not loaded.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38

5E03H (DOS 3.10 to 3.30)
Get Printer Setup

and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

IMPORTANT: The redirection index value may change if function call 5F03H (Redirect Device) or function call 5F04H (Cancel Redirection) is issued between the time the redirection list is scanned and function call 5E03H (Get printer setup) is issued. Therefore, we recommend that you issue "Get printer setup" immediately after you issue "Get redirection list."

5F02H (DOS 3.10 to 3.30)
Get Redirection List Entry

Purpose:

Returns nonlocal network assignments.

On Entry	Register Contents
AX	5F02H
BX	Redirection index (zero-based)
DS:DI	Pointer to a 128 – byte buffer address of the local device name
ES:DI	Pointer to a 128 – byte buffer address of network name

On Return	Register Contents
AX	Error codes if carry flag is set
BH	Device status flag Bit 0 = 0 if device is valid 0 = 1 if device is not valid Bits 1-7 are reserved
BL	Device type
CX	Stored parm value
DX	Destroyed
BP	Destroyed
DS:SI	ASCIIZ local device name
ES:DI	ASCIIZ network name

5F02H (DOS 3.10 to 3.30)
Get Redirection List Entry

Remarks:

The Get Redirection List Entry function call returns the list of network redirections that were created through function call 5F03H (Redirect Device). Each call returns one redirection, so BX should be incremented by 1 each time to step through the list. The contents of the list may change between calls. The end-of-list is detected by error code 18 (no more files). Error code 1 (Invalid function number) is returned if the IBM PC Local Area Network Program is not loaded.

If either disk or print redirection is paused, the function is not affected.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

5F03H (DOS 3.10 to 3.30)
Redirect Device

Purpose:

Causes a Redirector/Server connection to be made.

On Entry	Register Contents
AX	5F03H
BL	Device type 03 Printer device 04 File device
CX	Value to save for caller
DS:SI	Source ASCIIZ device name
ES:DI	Destination ASCIIZ network path with password

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

This call is the interface that defines the current directories for the network and defines redirection of network printers.

- If BL = 03, the source specifies a printer, the destination specifies a network path, and the CX register has a word that DOS maintains for the programmer. For compatibility with the IBM PC Local Area Network Program, CX should be set to 0. Values other than 0 are reserved for the

5F03H (DOS 3.10 to 3.30)
Redirect Device

IBM PC Local Area Network Program. This word may be retrieved through function call 5F02H (Get Redirection List). All output destined for the specified printer is buffered and sent to the remote printer spool for that device. The printers are redirected at the INT 17H level.

The source string must be **PRN** , **LPT1**, **LPT2**, or **LPT3**, each ended with a 00H. The destination string must point to a network name string of the following form:

[*computername*\{*shortname*|*printdevice*}]

The destination string must be ended with a 00H.

The ASCIIZ password (0 to 8 characters) for access to the remote device should immediately follow the network string. The password must end with a 00H. A null (zero length) password is considered to be no password.

- If BL = 4, the source specifies a drive letter and colon ended with 00H, the destination specifies a network path ended with 00H, and the CX register has a word that DOS maintains for the programmer. For compatibility with the IBM PC Local Area Network Program, CX should be set to 00H. Values other than 00H are reserved for the IBM PC Local Area Network Program. The value may be retrieved through function call 5F02H (Get Redirection List). If the source was a drive letter, the association is made between the drive letter and the network path. All subsequent references to the drive letter are translated to references to the network path. If the source is an empty string, the system attempts to grant access to the destination with the specified password without redirecting any device.

The ASCIIZ password for access to the remote path should immediately follow the network string. A null (zero length) password ended with 00H is considered to be no password.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Notes:

1. Devices redirected through this function call are not displayed by the NET USE command.

2. An error is returned if you try to redirect a file device while disk redirection is paused, or if you try to redirect a printer while print redirection is paused.

3. Only the application that redirects a device can get access to that device. No other application running under TopView can get access to that device.

4. An application running under TopView can only cancel a redirection that it created. It cannot cancel a redirection created by another application running under TopView.

5F04H (DOS 3.10 to 3.30)
Cancel Redirection

Purpose:

Cancels a previous redirection.

On Entry	Register Contents
AX	5F04H
DS:SI	ASCIIZ device name or path

On Return	Register Contents
AX	Error codes if carry flag is set

Remarks:

The redirection created by the Redirect Device function call (5F03H) is removed through the Cancel Redirection call. If the buffer points to a drive letter and the drive is associated with a network name, the association is terminated and the drive is restored to its physical meaning. If the buffer points to PRN, LPT1, LPT2, or LPT3, and the device has an association with a network device, the association is terminated and the device is restored to its physical meaning. If the buffer points to a network path ended with 00H and a password ended with 00H, then the association between the local machine and the network directory is terminated.

An error is returned if you try to cancel a redirected file device while disk redirection is paused, or if you try to cancel a redirected printer while print redirection is paused. Error code 1 (Invalid function

number) is returned if the IBM PC Local Area Network Program is not loaded.

Error codes are returned in AX. Issue function call 59H Get Extended Error for additional information about the error class, suggested action, and locus. Refer to "Error Return Information" on page 6-38 and "Extended Error Codes" on page 6-42 for more information on the codes returned from function call 59H.

Note: An application running under TopView can only cancel a redirection that it created. It cannot cancel a redirection created by another application running under TopView.

62H (DOS 3.00 to 3.30)
Get Program Segment Prefix Address

Purpose:

Returns the program prefix address.

On Entry	Register Contents
AH	62H

On Return	Register Contents
BX	Segment address of the currently executing process

Remarks:

The internal PSP address for the currently executing process is returned in BX.

65H (DOS 3.30) Get Extended Country Information

Purpose:

Returns extended country information

On Entry	Register Contents
AH	65H
AL	ID value of information of interest (1, 2, 4, 5, or 6)
BX	Code page of interest (-1 = active CON device)
DX	Country ID for which information is returned (default is -1)
CX	Amount of data to return
ES:DI	Buffer where country information is returned

On Return	Register Contents
ES:DI	Extended country information table

Remarks:

On entry, DX contains the ID of the country for which the extended information is needed. AL contains the ID value for the country.

- If the country code and code page do not match, or either or both are invalid, an error code of 2 (file not found) is returned in AX.

65H (DOS 3.30) Get Extended Country Information

- The size requested in CX must be 5 or greater. If it is less than 5, an error code of 1 is returned in AX.

- If the amount of information returned is greater than the size requested in CX, it is truncated and no error is returned in AX.

The following tables show what information is returned for each valid country Info id value:

Country Information	Size
Info id = 01	1 byte
size (38 or less)	2 bytes
Country id	2 bytes
Code page	2 bytes
Date format	2 bytes
Currency symbol	5 bytes
1000 separator	2 bytes
Decimal separator	2 bytes
Date separator	2 bytes
Time separator	2 bytes
Currency format flags	1 byte
Digits in currency	1 byte
Time format	1 byte
Monocase routine entry point	4 bytes
Data list separator	2 bytes
Zeros	10 bytes

Note: For further information on the country information, see function call 38H for DOS 3.00 to 3.30.

Uppercase Table	Size
Info id = 02	1 byte
Doubleword pointer to uppercase table	4 bytes

File name Uppercase Table	Size
Info id = 04	1 byte
Doubleword pointer to filename uppercase table	4 bytes

The uppercase table and the filename uppercase tables are 130 bytes long consisting of a length field (2 bytes) followed by 128 uppercase values for the upper 128 ASCII characters. They have the following layout:

- bytes 0 and 1 = size of table
- bytes 2 - 129 = list of uppercase values

The following formula can be used to determine the address of an uppercase equivalent for a lowercase character (ASCII_in) in the uppercase table or the filename uppercase table.

```
ASCII_in -(256-table_len)+
table_start
= address of ASCII_out
```

Where:

```
ASCII_in     = character to be generated

table_len    = length of list of uppercase
values (2 bytes)
```

65H (DOS 3.30) Get Extended Country Information

```
table_start = starting address of uppercase
table (4 bytes)

ASCII_out    = uppercase value for ASCII_in
```

If the value of ASCII_in is equal to or greater than (256-table_len) there is an uppercase equivalent for ASCII_in in the table. If it is lower than (256-table_len) no uppercase equivalent exists in the table.

Collate Size	Table
Info id = 06	1 byte
Doubleword pointer to collating sequence	4 bytes

The collate table is 258 bytes long, consisting of a length field (two bytes) followed by 256 ASCII values, in the appropriate order.

66H (DOS 3.30) Get/Set Global Code Page

Purpose:

This function changes the code page for the current country.

On Entry	Register Contents
AH	66H
AL	01 - Get Global Code Page

On Return	Register Contents
AX	Error code if carry flag set.
BX	Active code page (currently set by user)
DX	System code page (active code page at boot time)

On Entry	Register Contents
AH	66H
AL	02 - Set Global Code Page

66H (DOS 3.30) Get/Set Global Code Page

On Return	Register Contents
AX	Error code if carry set.

Remarks:

DOS moves the new code page data from the COUNTRY.SYS file to a resident country buffer area. DOS uses the new code page to perform a Select to all attached devices that are set up for code page swicthing (have a code page switching device driver specified in CONFIG.SYS). If any device fails to be selected, an error code of 65 is returned in AX. The code page must be recognizable by the the current country and DOS must be able to open and read from the country information file, otherwise, the carry flag will be set on return and AX will contain 02 (file not found).

Notes:

1. The only way to change the code page used by the current country is to reboot the system with a different CONFIG.SYS.

2. NLSFUNC must be installed to use this function call, and all the devices must be prepared in order for the Select function to be successful.

67H (DOS 3.30) Set Handle Count

Purpose:

Permits more than 20 open files per process.

On Entry	Register Contents
AX	67H
BX	Number of open handles allowed

On Return	Register Contents
AX	Error code if carry set.

Remarks:

The maximum number of file handles allowed for this interrupt is 64K. If the the specified number of allowable handles is less than the current number allowed, the specified number will only become current after all the handles above the number being specified have been closed. If the specified number is less than 20, the number is assumed to be 20. Values of up to 255 will be allowed in the CONFIG.SYS command FILES= . Data base applications can use this function to reduce the need to swap handles.

You must release memory for DOS to contain the extended handle list. You can do this by using the SET BLOCK (4AH) function call.

68H (DOS 3.30) Commit File

Purpose:

This function call causes all buffered data for a file to be written to the device. This function can be used instead of the close-open sequence.

On Entry	Register Contents
AX	68H

On Return	Register Contents
BX	File handle

Remarks:

Commit File provides a faster and more secure method of committing data in multi-user environments such as the IBM PC Local Area Network.

Chapter 7. DOS Control Blocks and Work Areas

Introduction

This chapter contains:

- A description of the locations and usage of the DOS memory map.
- A detailed description and diagram of the program segment prefix.
- A detailed description and diagram of the file control block (standard and extended).

DOS Memory Map

Location	Usage
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 22H (terminate), 23H (Ctrl-Break), 24H (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
XXXX:0000	Transient portion of COMMAND.COM

Notes:

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

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 call 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 started. This area is called the Program Segment.

At offset 0 within the Program Segment, DOS builds the Program Segment Prefix control block. EXEC loads the program at offset 100H and gives it control.

The program returns from EXEC by a jump to offset 0 in the Program Segment Prefix, by issuing an INT 20H, by issuing an INT 21H with register AH = 00H or 4CH, or by calling location 50H in the Program Segment Prefix with AH = 00H or 4CH.

> **Note:** It is the responsibility of all programs to ensure that the CS register contains the segment address of the Program Segment Prefix when terminating using any of these methods except call 4CH.

All of these methods result in returning to the program that issued the EXEC. During this returning process, interrupt vectors 22H, 23H, and 24H (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.

When a program receives control, the following conditions are in effect:

For all programs:

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

  The environment is a series of ASCII strings (totaling less than 32K bytes) 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. Following the byte of zeros that terminates the set of environment strings is a set of initial arguments passed to a program that contains a word count followed by an ASCIIZ string. The ASCIIZ string contains the drive, path, and *filename{.ext}* of the executable program. Programs may use this area to determine where the program was loaded from. The environment built by the command processor (and passed to all programs it invokes) contains a COMSPEC= *string* at a minimum (the parameter on COMSPEC is the path used by DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings entered through the SET command. See Chapter 7 of the *DOS Reference* for more information on the PATH and PROMPT commands.

  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, it will not change even if subsequent SET, PATH, or PROMPT commands are issued.

- Offset 50H 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 + 50H to invoke a DOS function, rather than issuing an interrupt type 21H.

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

- File control blocks at 5CH and 6CH are formatted from the first two parameters 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 81H contains all the characters entered after the command name (including leading and imbedded delimiters), with 80H 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.

- For .COM files, 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 = FFH if the first parameter contained an invalid drive specifier (otherwise AL = 00H)

  - AH = FFH if the second parameter contained an invalid drive specifier (otherwise AH = 00H)

For .EXE programs:

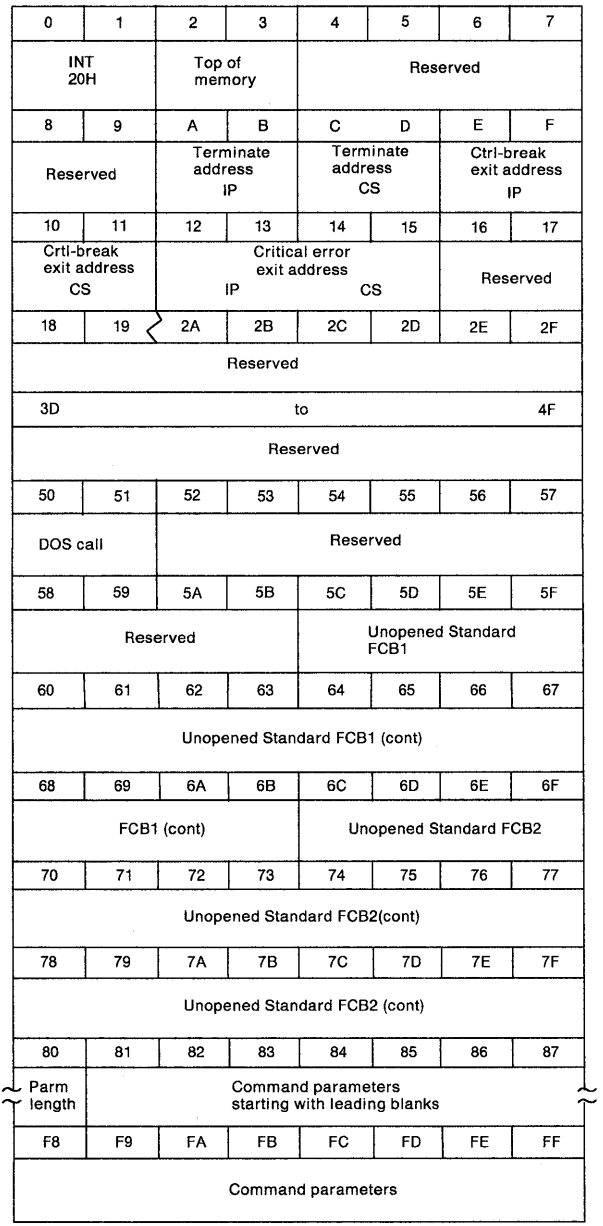
- DS and ES registers are set to point to the Program Segment Prefix.
- CS, IP, SS, and SP registers are set to the values passed by the Linker.

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 (4AH) function call, to provide space for the program being invoked.
- The Instruction Pointer (IP) is set to 100H.
- SP register is set to the end of the program's segment. The segment size at offset 6 is rounded down to the paragraph 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



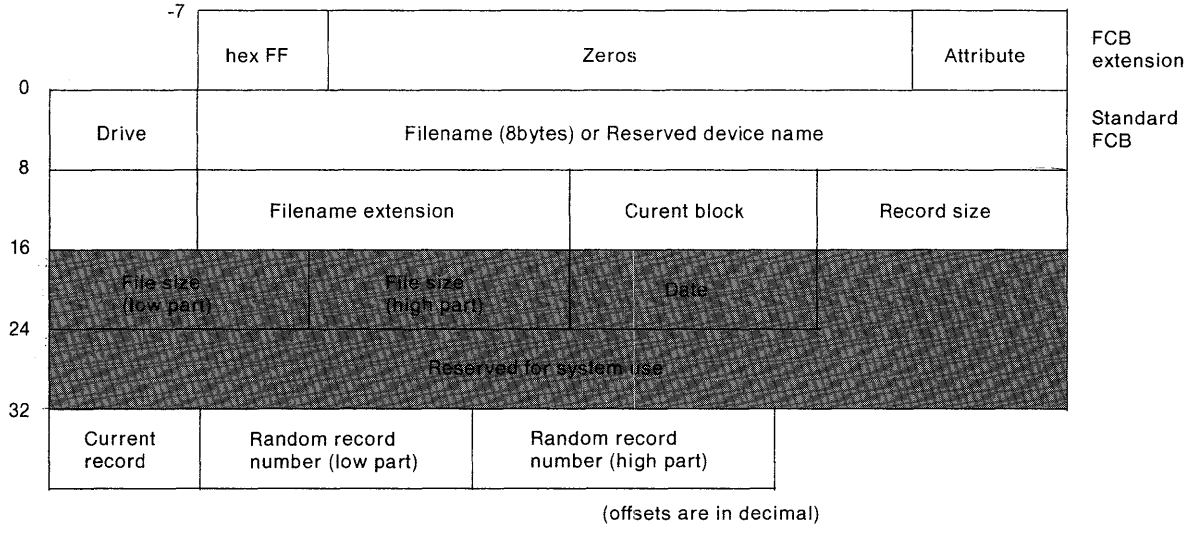
1. First segment of available memory is in segment (paragraph) form (for example, 1000H would represent 64K).

2. The word at offset 6 contains the number of bytes available in the segment.

3. Offset 2CH contains the segment address of the environment.

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

ENVIRONMENT STRING SPACE
DOS 3.3 = E08
3.1 = D11
3.0 = F2C



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: 0 - drive A 1 - drive A 2 - drive B etc. 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 0 (set to 0 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 80H 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 0. 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 3 bytes are used. Note that if you use the File Control Block at 5CH 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 80H at offset 14, and 00H 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 FFH to indicate an extended FCB.
FCB-6 to FCB-2	Reserved.
FCB-1	Attribute byte. See "DOS Disk Directory" on page 5-10 of this book for the attribute bit definitions. Also refer to function call 11H (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 reference in the DOS Function Calls (refer to Chapter 6 of this book) to an FCB, whether opened or unopened, may use either a normal or extended FCB. If you are using an extended FCB, the appropriate register should be set to the first byte of the prefix, rather than the drive-number field.

Font Files

Code page images are contained in font files. There are two display font files (EGA.CPI and LCD.CPI), and two printer font files (4201.CPI and 5202.CPI). These font files contain the character images of the code pages, which can be down loaded to the device, if it has a code page switching device driver entry in CONFIG.SYS

Font files have the following layout:

```
head:

  db    0ffh,"font    " ;file tag(7 characters)
        db    8 dup(0)  ;reserved
  dw    1               ;number of pointers in header,
                        ;should be 1 for DOS 3.30
  db    1               ;type of info pointer,
                        ;should be 1 for DOS 3.30
  dw    offset info,0   ;displacement of info from
                        ;start of file (2 words)
info:

  dw    1               ;number of code page entries

cphead:
  dw    len_cphead              ;size of this code
                        ;page entry header
  dw    0, 0            ;pointer to header
                        ;of next code page entry
                        ;(0, 0 for last header)
  dw    1               ;device type,
                        ;1 = display 2 = printer
  db    "EGA      "     ;device sub type id (8 bytes)
                        ;name of font file
                        ;(EGA, LCD, 4201 or 5202)
  dw    999             ;code page id
  dw    3 dup(0)        ;reserved
  dw    offset dathead,0    ;pointer to fonts
                        ;(two word value)
len_cphead equ  ($-cphead)

dathead:

  dw    1               ;reserved (must be 1)
```

```
  dw    3                ;number of fonts
  dw    len_data                  ;length of following
;font data
```

The layout of the data portion of a font file is depends on whether the device is a display or a printer. The next three examples show the font data layouts for displays and printers.

Font data for displays:

```
fnthead_16:

  db    16, 8            ;character box size
                         ;(rows, columns)
  db    0,0              ;aspect ratio (unused)
  dw    256              ;number of characters


                         .
                         .
                         .

          pixel description for all 256
          characters within the code page

                         .
                         .
                         .


fnthead_14:
  db    14, 8            ;character box size
  db    0, 0             ;aspect ratio (unused)
  dw    256              ;number of characters

                         .
                         .
                         .

pixel description for all 256
characters within the code page

                         .
                         .
                         .

fnthead_8:

  db    8, 8             ;character box size
```

```
  db    0, 0              ;aspect ratio (unused)
  dw    256               ;number of characters

                          .
                          .
                          .

pixel description for all 256
characters within the code page

                          .
                          .
                          .


len_data  equ ($ - fnthead_16)
```

Font data for Proprinter with RAM font loading option.

```
sel_type:

  dw  1                   ;selection type:
                          ;1=4201, 2=5202
  dw  12                  ;total number of bytes in
                          ;the control sequences
  db  5,27,"I",0,27,"6"   ;define hardware code page
                          ;(max length of 31)
  db  5,27,"I",4,27,"6"   ;define downloadable code
                          ;page
                          ;(max length of 31)

                        .
                        .
                        .
Character description for download to printer,
see Proprinter Technical Reference for details
                        .
                        .
                        .

len_data equ  ($ - sel_type)
```

Font data for Quietwriter III

```
sel_type

  dw  2              ;selection type:
                     ;1=4201, 2=5202
  dw  12             ;total number of
                     ;bytes in the control
                     ;sequences
  db  27,"[","T",5,0,0,0,1,181,0,27,"6"
                     ;select active
                     ;code page 437
                     ;(max length of 31)

                       .
                       .
                       .
Character description for download to
printer, see Quietwriter III Technical
Reference for details.
                       .
                       .
                       .
len_data equ ($ - sel_type)
```

Chapter 8. Executing Commands from within an Application

Introduction

Application programs may invoke a secondary copy of the command processor. Your program may pass a DOS command as a parameter that the secondary command processor will execute as though it had been entered from the standard input device.

Invoking a Command Processor

The procedure is:

1. Assure that adequate free memory (17K bytes for DOS versions 2.10 and 3.20, and 23K bytes for DOS versions 3.10 to 3.30) exists to contain the second copy of the command processor and the command it is to execute. This is accomplished by executing function call 4AH to shrink memory allocated to that of your current requirements. Next, execute function call 48H with BX=FFFFH. This returns with the amount of memory available.

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 = 0DH (carriage return)

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

```
DB 19, "/C C:DISKCOPY A: B:" , 13
```

3. Use the EXEC function call (4BH, 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 gotten from the COMSPEC = *parameter* in the environment passed to you at PSP + 2CH). Remember to set offset 2 of the EXEC control block to point to the string built above.

Chapter 9. Fixed Disk Information

Introduction

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 backup/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" one or more partitions. The number and sizes of the partitions is user-selectable through a fixed disk utility program. The DOS utility is FDISK.COM. 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. With the DOS FDISK utility, you may designate a partition as "bootable" (active) - the master fixed disk boot record causes 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 cannot be successfully read within five retries 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," the message **Missing operating system** appears, and the system enters an enabled loop. See "Boot Record Partition Table" on page 9-6 for complete information about the boot record.

   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 or logical drives within an extended partition and contains:

1. Code to load and give control to the boot record for one of four 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 four 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. FDISK initialization utilities mark a user-selected partition as "bootable" by placing a value of 80H in the corresponding partition's boot indicator (setting all other partition's indicators to 0 at the same time). The presence of the 80H tells the standard boot routine to load the sector whose location is contained in the following 3 bytes. That sector is the actual boot record for the selected operating system, and it is 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:

Offs	Purpose	Head		Sector	Cylinder
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 sect	Low word		High word	
1FA	Partition 4 # sects	Low word		High word	
1FE	Signature				

Boot Indicator (Boot Ind): The boot indicator byte must contain 0 for a non-bootable partition, or 80H for a bootable partition. Only one partition can be marked bootable.

System Indicator (Sys Ind): The "syst ind" field contains an indicator of the operating system that "owns" the partition or points to the extended partition.

The system indicators for DOS are:

00H - unknown (unspecified)

01H - PrimaryDOS 12-bit FAT

04H - Primary DOS 16-bit FAT

05H - Extended DOS

Cylinder (CYL) and Sector (S): 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 13H (Disk I/O) requirements, to allow for a 10-bit cylinder number.

The fields are ordered in such a manner that only two 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 (80H) corresponds to the boot indicator byte).

All partitions are allocated in cylinder multiples and begin on sector 1, head 0.

EXCEPTION: The partition that is allocated at the beginning of the disk should start at cylinder 0, head 1, sector 1, to leave room for the disk's master boot record and other information used to define the fixed disk type on that system. An operating system should not use any data space on cylinder 0, head 0 of a fixed disk.

Relative Sector (Rel Sect): 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.

Number of Sectors (# Sects): 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.

Signature: The last 2 bytes of the boot record (55AAH) 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 1FEH.

The master disk boot record invokes 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) 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.

4. Such utilities should not change or move any partition information that belongs to another operating system.

Extended DOS Partition

The extended DOS partition is a new partition type intended to allow future DOS expansion on a large fixed DASD. The extended partition will be indicated by a system indicator byte of 05H in the partition table of the Master Boot Record. This partition is not bootable, and programs that can set bootable partitions (such as DOS FDISK) should not allow the partition to be marked as bootable.

The extended DOS partition can only be created if a primary DOS partition already exists on a bootable drive. A primary DOS partition is a partition with a system id byte of 01H or 04H. If the drive is not bootable, then an extended DOS partition may be created without having a primary DOS partition.

The extended DOS Partition starts and ends on a cylinder boundary.

Extended DOS Partition Architecture

The extended DOS Partition is a collection of extended volumes that are linked together by a pointer in the extended volume's extended boot record. An extended volume consists of an extended boot record and one logical block device.

An extended volume created within the extended DOS partition can be any size from 1 cylinder long up to the maximum available contiguous space in the extended DOS partition. However, in DOS 3.30 an extended volume cannot be larger than 32 MB due to the limitations of the FAT file system. All extended volumes must start and end on a cylinder boundary. An extended volume will correspond to an image of a physical disk. The extended boot record

corresponds to the master boot record at the beginning of an actual physical disk, and the logical block device corresponds to the DOS partition that is pointed to by the master boot record.

Therefore, the logical block device begins with a normal DOS boot sector if it is a DOS logical block device (syst ind = 01H or 04H). This logical block device must start on a track boundary and follows the extended boot record on the physical disk. The logical block device and the extended volume both end on the same cylinder boundary.

Extended Partition Boot Record

Each extended volume contains an extended boot record in the first sector of the disk location assigned to it. This boot record contains the 55AAH signature id byte. This allows programs that look at the extended (master) boot record to be compatible. This boot record also contains a logical drive table which can contain only two types of entries. The boot code is not critical because the devices are not considered bootable. It is suggested that the boot code simply output a message indicating an unbootable partition if it is executed.

The boot record for the extended DOS partition is similar to the master boot record. The primary differences are that the boot record for the extended partition contains a drive table instead of a partition table, and a system indicator value of 05H points to the next logical drive rather than to another extended partition.

Extended Partition Boot Record Logical Drive Table

The logical drive table portion of the extended boot record is the same as the partition table structure in the master boot record. This structure has four entries of 16 bytes each. The system indicator byte must be filled in for all 4 entries with one of the following values:

00H - No space allocated in this entry.

01H - DOS partition with 12-bit FAT

04H - DOS partition with 16-bit FAT

05H - Maps out area assigned to the next extended volume and serves as a pointer to the next extended boot record.

06H - Reserved for future use.

If the system indicator byte is 0, the values in that partition table entry should be zeros.

The drive start and end fields (C,H,S) should be filled in for any of the 4 logical drive entries in an extended boot record that have one of the above system id bytes. This allows a program such as FDISK to determine the allocated space in the extended DOS partition, and allows the device drivers to determine the physical DASD area that belongs to it.

The drive start and end fields (C,H,S) for the partition entry that points to the logical block device (system id 01H, 04H, or 06H) map out the physical boundaries of the logical block device, and they are offset relative to the bginning of the extended boot record that the entry resides in. The drive start and

end fields (C,H,S) for the drive entry that points to the next extended volume (system id 05H) map out the physical boundaries of the next extended volume and they are relative to the beginning of the entire physical disk.

The relative sector and number of sector fields will be set up differently depending on which system id byte is used. If 01H, 04H, or 06H are in the system id field for that drive entry (pointer to the logical block device), the relative sector field should be set up as an offset from (and including) the start of the extended boot record for the associated extended volume. The number of sectors (size) field will be filled in with the size of the created logical block device area (or in other words, the number of sectors mapped out by the start and stop cylinder/track/sector fields). The size of the extended volume can be calculated by adding the relative sector field and the sector size field of the associated extended boot record.

If the system id byte is 05H, the relative sector field is offset (of the NEXT extended volume) in sectors from the start of the entire extended DOS partition. The number of sectors field is not used in this field, and should be filled with 00H's.

This architecture allows only one logical block device to be defined per extended boot record. Therefore, a maximum of only two partition entries at a time will be used in each extended boot record: an entry with system id byte of 01H, 04H, or 06H and an entry with a system id of 05H, which is the pointer to the next extended volume. Although only two entries can be used, a program installing these devices should not assume that the first two entries will be the non-zero entries.

The last two bytes of the extended boot record (55AAH) are used as a signature to identify a valid boot record. Both this record and the logical drive

boot records are required to contain the signature at offset 1FEH.

The logical drive table with its offsets into the boot record is:

Offs	Purpose	Head	Sector	Cylinder
1BE	Drive begin	boot ind	H	S	CYL
1C2	Drive end	syst ind	H	S	CYL
1C6	Drive rel sect	Low word	High word
1CA	Drive # sects	Low word	High word
1CE	Drive begin	boot ind	H	S	CYL
1D2	Drive end	syst ind	H	S	CYL
1D6	Drive rel sect	Low word	High word
1DA	Drive # sects	Low word	High word
1DE	Drive begin	boot ind	H	S	CYL
1E2	Drive end	syst ind	H	S	CYL
1E6	Drive rel sect	Low word	High word
1EA	Drive # sects	Low word	High word
1EE	Drive begin	boot ind	H	S	CYL
1F2	Drive end	syst ind	H	S	CYL
1F6	Drive rel sect	Low word	High word
1FA	Drive # sects	Low word	High word
1FE	Signature	

Determining Fixed Disk Allocation

DOS determines disk allocation using the following formula:

$$SPF = \frac{TS - RS - \frac{D * BPD}{BPS}}{CF + \frac{BPS * SPC}{BPC}}$$

The parameters are:

TS The count of the total sectors on the disk.

RS The number of sectors at the beginning of the disk that are reserved for the boot record. DOS reserves 1 sector.

D The number of directory entries in the root directory. Refer to "DOS Disk Directory" in chapter 5 for more information

BPD The number of bytes per directory entry. BPB is always 32.

BPS The number of bytes per logical sector. Typically, BPS is 512, but you can specify a different value using VDISK.

CF The number of FATs per disk. For most disks CF is 2. For VDISK CF is 1.

SPF The number of sectors per FAT. The maximum value for SPF is 64.

SPC The number of sectors per allocation unit.

BPC The number of bytes per FAT entry. BPC is 1.5 for 12-bit FATs and 2 for 16-bit FATS.

Chapter 10. EXE File Structure and Loading

Introduction

This chapter contains information on:

- The .EXE file structure
- The relocation table

.EXE File Structure

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 begins in the memory image of the module constructed by the Linker.

The header is formatted as follows:

Note: Use the value at hex offset 18 – 19 to located the first entry in the relocation table.

Hex	Offset Contents
00-01	4DH, 5AH—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 .
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.
0C-0D	Maximum number of 16-byte paragraphs required above the end of the loaded program.
0E-0F	Displacement in paragraphs of stack segment within load module.
10-11	Offset 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	Offset to be in the IP register when the module is given control.
16-17	Displacement in paragraphs of code segment within load module.
18-19	Displacement in bytes of the first relocation item within the file.
1A-1B	Overlay number (0 for resident part of the program).

The Relocation Table

The word at 18H locates the first entry in the relocation table. 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 represent the displacement 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 (its 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 Link programs prior to version 1.10 *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.

   Note: The relocation table is an unordered list of relocation items. The first relocation item is the one that has the lowest offset in the file.

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 11. DOS Memory Management

Introduction

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

Control Block

DOS manages memory as follows:

DOS builds a control block for each block of memory, whether free or allocated. For example, if a program issues an "allocate," 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 you do a setblock 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 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 corrective action is to restart the system).

When a program (command or application program) is to be loaded, DOS uses the EXEC function call (4BH) 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).

- 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.

Function 0: 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. 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.

> **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 0CH, if that much memory is available. If not, EXEC allocates the size of the program's memory image plus the value in the MIN ALLOC field in the header (offset 0AH). These fields are set by the Linker.

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.

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

When a program loaded using EXEC function 0 exits, its initial allocation block (the block beginning with its PSP) is automatically freed before the calling program regains control. It is the responsibility of all programs to free any memory they allocate, before exiting to the calling program.

Function 3: 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 that will be loaded using the "load overlay" call. Note that "load overlay" does not check to see if the calling program actually owns 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 chance that the program being loaded will overlay one of the control blocks that DOS uses to keep track of memory blocks.

Programs loaded using function 3 should *not* issue any SETBLOCK calls, since they don't own the

memory they are operating in (the memory is owned by the *calling* program).

Because programs loaded using 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.

Section 2

Chapter 12. The Linker (LINK) and EXE2BIN Programs

Introduction

The linker (LINK) program:

- Combines separately produced object modules
- Searches library files for definitions of unresolved external references
- Resolves external cross-references
- Produces a printable listing that shows the resolution of external references and error messages
- Produces a relocatable load module.

The LINK program resides on your DOS Utilities Program Diskette. This chapter shows you how to use LINK. Read all of this chapter before you start LINK.

Files

The linker processes input, output, and temporary files.

Input Files

Input Files Used by the Linker

Type	Default .ext	Override .ext	Produced by
Object	OBJ	Yes	Compiler[1] or MACRO Assembler
Library	LIB	Yes	Compiler and user
Automatic Response	(None)	N/A*	User

*N/A - Not applicable.

[1] One of the optional compiler packages available for use with the IBM Personal Computer DOS.

Output Files

Output Files Created by the Linker

Type	Default .ext	Override .ext	Used by
Listing	.MAP	Yes	User
Run	.EXE	No	Relocatable loader (COMMAND.COM)

VM.TMP (Temporary File)

LINK uses as much memory as is available to hold the data that defines the load module being created. If the module is too large to be processed with the available amount of memory, the linker may need additional memory space. If this happens, a temporary file called VM.TMP is created on the DOS default drive.

When the overflow to the VM.TMP file has begun, the linker displays the following message:

```
VM.TMP has been created
Do not change diskette in drive x
```

If the VM.TMP file has been created on diskette, you should not remove the diskette until LINK ends. When LINK ends, it erases the VM.TMP file.

If the DOS default drive already has a file by the name of VM.TMP, it is deleted by LINK and a new file is allocated; the contents of the previous file are destroyed. Therefore, you should avoid using VM.TMP as one of your own file names.

Definitions

Segment, *group*, and *class* are terms that appear in this chapter and in some of the messages in Appendix A of the *DOS Reference* These terms describe the underlying function of LINK.

Segment

A *segment* is a contiguous area of memory up to 64K bytes in length. A segment may be located anywhere in memory on a *paragraph* (16-byte) boundary. Each of the four segment registers defines a segment. The segments can overlap. The contents of a segment are addressed by a segment register/offset pair.

The contents of various portions of the segment are determined when machine language is generated.

Neither size nor location is necessarily fixed by the compiler or assembler because this portion of the segment may be combined at link time with other portions forming a single segment.

A program's ultimate location in memory is determined at load time by the relocation loader facility provided in COMMAND.COM, based on whether you specified the /HIGH parameter. The /HIGH parameter is discussed later in this chapter.

Group

A *group* is a collection of segments that fit together within a 64K byte segment of memory. The segments are named to the group by the assembler or compiler. A program may consist of one or more groups.

The group is used for addressing segments in memory. The various portions of segments within the group are addressed by a segment base pointer plus an offset.

Class

A *class* is a collection of segments. The naming of segments to a class affects the order and relative placement of segments in memory. The class name is specified by the assembler or compiler. All portions assigned to the same class name are loaded into memory contiguously.

The segments are ordered within a class in the order that the linker encounters the segments in the object files. One class precedes another in memory only if a segment for the first class precedes all segments for the second class in the input to LINK. Classes are not restricted in size.

Command Prompts

After you start the linker session, you receive a series of four prompts. You can respond to these prompts from the keyboard, on the command line, or by using a special diskette file called an *automatic response file.* An example of an automatic response file is provided in this chapter.

LINK prompts you for the names of the object, run, list, and library files. When the session is finished, LINK returns to DOS and the DOS prompt is displayed. If linking is unsuccessful, LINK displays a message.

The prompts are described in order of their appearance on the screen. Defaults are shown in square brackets ([ ]) after the prompt. In the response column of the table, square brackets indicate optional entries. **Object Modules** is the only prompt that requires a response from you.

PROMPT	**RESPONSE**
Object Modules .OBJ :	[*d:*][path]*filename*[*.ext*] [+.[*d:*][path]*filename* [*.ext*]]...
Run File filename.EXE :	[*d:*][path][*filename*[*.ext*]]
List File NUL.MAP :	[*d:*][path][*filename*[*.ext*]]
Libraries .LIB :	[*d:*][[path]*filename*[*.ext*]] [+.[*d:*][[path]*filename* [*.ext*]]]...

Notes:

1. If you enter a file name without specifying the drive, the default drive is assumed. If you enter a file name without specifying the path, the default path is assumed. The libraries prompt is an exception – the linker will look for the libraries on the default drive and if not found, look on the drive specified by the compiler.

2. You can end the linker session prior to its normal end by pressing Ctrl – Break.

Command Prompts

The following descriptions contain information about the responses that you can enter to the prompts.

Object Modules .OBJ :

Enter one or more file locations for the object modules to be linked. Multiple file locations must be separated by single plus (+) signs or blanks. If the extension is omitted from any file name, LINK assumes the file name extension **.OBJ**. If an object module has a different file name extension, the extension must be specified. Object file names can not begin with the @ symbol (@ is reserved for using an automatic response file).

LINK loads segments into classes in the order in which they are encountered.

If you specify an object module on a diskette drive, but LINK cannot locate the file, it displays the following prompt:

```
Cannot find file object module
change diskette <press ENTER>
```

If you specify an object module on a non-removable media (like a fixed disk), the linker session ends with the following message:

```
Cannot find file object module
```

You should insert the diskette containing the requested module. This permits **.OBJ** files from several diskettes to be included. On a single-drive system, diskette exchanging can be done safely *only* if VM.TMP has *not* been opened. As explained in the discussion of the VM.TMP file earlier in this chapter, a message will indicate if VM.TMP has been opened.

Important: If a VM.TMP file has been opened on a diskette, you should *not* remove the diskette containing the VM.TMP file.

After a VM.TMP file has been opened, if you specified an object module on the same disk that VM.TMP is on and LINK cannot find it, the linker session ends with the message:

```
Cannot find file object module
```

Run File filename.EXE :

The file specification you enter is created to store the run (executable) file that results from the LINK session. All run files receive the file name extension **.EXE**, even if you specify another extension. If you specify another extension, it is ignored.

The default file name for the run file prompt is the first file name specified on the object module prompt.

You can specify just a drive letter, or a path on the run file prompt. This changes the place where the run file *filename*.EXE is placed.

List File NUL.MAP :

The linker list file is sometimes called the linker *map*.

The list file contains an entry for each segment in the input (object) modules. Each entry also shows the offset (addressing) in the run file.

The list file is not created unless you specifically request it. You can request it by overriding the default with a drive letter, path, or *filename*[*.ext*]. If you do not include a file name extension, the default extension **.MAP** is used. If you do not enter anything, the DOS reserved file name **NUL** specifies that no list file is created.

You can specify just a drive letter or a path on the list file prompt. This changes the place where the list file is placed.

Important: If the list file is allocated to a file on diskette, that diskette must not be removed until the LINK has ended.

> **Note:** There is one exception. If /P is specified, the diskette containing the list file may be removed while the .EXE file is being written. The linker prompts you to put back the diskette containing the list file when it finishes writing the .EXE file.

If you specify an object module on the same diskette drive as the diskette drive to which the list file is allocated, and LINK cannot find the object module, the linker session ends with the message:

```
Cannot find file object module
```

To avoid generating the list file on a diskette, you can specify the display or printer as the list file device. For example:

```
List File [NUL.MAP]: CON
```

If you direct the output to your display, you can also print a copy of the output by pressing Ctrl-PrtSc.

Libraries .LIB :

You may either list the file locations for your libraries, or just press the Enter key. If you press the Enter key, LINK defaults to the library provided as part of the Compiler package.

The LINK program looks for the Compiler package library on the default drive. If it cannot find the library there, it looks for the library on the drive specified by the Compiler package. For linking objects from just the MACRO Assembler, there is no automatic default library search.

If you answer the library prompt, you specify a list of drive letters and [*path*]*filename.ext* separated by plus signs (+) or spaces. You can enter from 1 to 16 library file locations. Specifying a drive letter tells linker to look on that drive instead of the Compiler package supplied drive for all subsequent libraries on the library prompt. The automatically searched library file specifications are conceptually placed at the end of the response to the library prompt.

LINK searches the library files in the order they are listed to resolve external references. When LINK finds the module that defines the external symbol, the module is processed as another object module.

If two or more libraries have the same file name, regardless of the location, only the first library in the list is searched.

When LINK cannot find a library file, it displays a message like this:

```
Cannot find library A:library file
Enter new drive letter:
```

The drive that the indicated library is located on must be entered.

The following library prompt responses may be used:

Libraries [.LIB]: B:

Look for compiler.LIB on drive B.

Libraries [.LIB]: B:USERLIB

Look for USERLIB.LIB on drive B and compiler.LIB on drive A.

Libraries [.LIB]: A:LIB1 + LIB2 + B:LIB3 + A:

Look for LIB1.LIB and LIB2.LIB on drive A, LIB3.LIB on drive B, and compiler.LIB on drive A.

Linker Parameters

At the end of any of the four linker prompts, you may specify one or more parameters that instruct the linker to do something differently. Only the / and first letter of any parameter are required.

/DSALLOCATION

The /DSALLOCATION (/D) parameter directs LINK to load all data defined to be in DGROUP at the *high end* of the group. If the /HIGH parameter is specified, (module loaded high), any available storage below the specifically allocated area within DGROUP is allocated dynamically by your application. It still is addressable by the same data space pointer.

> **Note:** The maximum amount of storage which can be dynamically allocated by the application is 64K-bytes (or the amount actually available) minus the allocated portion of DGROUP.

If the /DSALLOCATION parameter is not specified, LINK loads all data defined to be in the group whose group name is DGROUP at the *low end* of the group, beginning at an offset of 0. The only storage thus referenced by the data space pointer should be that specifically defined as residing in the group.

All other segments of any type in any GROUP other than DGROUP are loaded at the low end of their respective groups, as if the /DSALLOCATION parameter were not specified.

For certain compiler packages, /DSALLOCATION is automatically used.

/HIGH

The /HIGH (/H) parameter causes the loader to place the run image as high as possible in storage. If you specify the /HIGH parameter, you tell the linker to cause the loader to place the run file as high as possible without overlaying the transient portion of COMMAND.COM, which occupies the highest area of storage when loaded. If you do not specify the /HIGH parameter, the linker directs the loader to place the run file as low in memory as possible.

The /HIGH parameter is used with the /DSALLOCATION parameter.

/LINE

For certain IBM Personal Computer language processors, the /LINE (/L) parameter directs LINK to include the line numbers and addresses of the source statements in the input modules in the list file.

/MAP

The /MAP (/M) parameter directs LINK to list all public (global) symbols defined in the input modules. For each symbol, LINK lists its value and segment-offset location in the run file. The symbols are listed at the end of the list file.

/PAUSE

The /PAUSE (/P) parameter tells LINK to display a message to you as follows:

```
About to generate .EXE file
Change diskette in drive X: and press <ENTER>
```

This message allows you to insert the diskette that is to contain the run file.

/STACK:size

The *size* entry is any positive decimal value up to 65536 bytes. This value is used to override the size of the stack that the MACRO Assembler or compiler has provided for the load module being created. If the size of the stack is too small, the results of executing the resulting load module are unpredictable.

If you do not specify /STACK (/S), the original stack size provided by the MACRO Assembler or compiler is used. This parameter can be used to reduce the stack size provided by the application only if the original stack has uninitialized data. In any case, it may increase the stack up to the 64K limit.

If the stack size is an odd number, either on the /S parameter or in the application's definition of the stack, LINK subtracts 1 to force the stack words to be on an even boundary for better efficiency when running on the 80286 processor.

At least one input (object) module must contain a stack allocation statement, unless you plan to use the EXE2BIN program. The stack allocation is automatically provided by compilers. For the MACRO Assembler, the source must contain a SEGMENT command that has the combine type of STACK. If a stack allocation statement was not provided, LINK returns the message **Warning: No Stack statement**.

/X

Use the /X parameter at runtime to adjust the total number of segments that an .EXE file can contain. You can vary the limit from 0 to 1024 segments. The default is 256 segments. This limit represents the number of distinct segments from all sources (object files and libraries) that the .EXE may contain.

Although the limit on the total number of segments may be set as high as 1024, the limit on the total number of segments that are not *absolute segments* is 1000. For a definition of *absolute segments*, see the assembler manual.

/O

To link object modules created by version 1 of the Pascal compiler or version 1 of the FORTRAN compiler using the 2.30 linker, specify the /O (old) switch.

How to Start the Linker Program

Before You Begin

- Make sure the files you use for linking are on the appropriate disks.
- Make sure you have enough free space on your disks to contain your files and any generated data.

You can start the linker program by using one of three options:

Option 1 - Console Responses

From your keyboard, type:

```
LINK
```

The linker is loaded into memory and displays a series of four prompts, one at a time, to which you must enter the requested responses. (Detailed

descriptions of the responses that you can make to the prompts are discussed in this chapter.)

If you enter a wrong response, such as an incorrectly spelled file name, you must press Ctrl-Break to exit LINK, then restart LINK. If the response in error has been typed but you haven't pressed Enter yet, you can delete the wrong characters (on that line only).

An example of a linker session using the console response option is provided in this chapter in the section called "How to Start the Linker Program."

As soon as you have entered the last file name, the linker begins to run. If the linker finds any errors, it displays the errors on the screen as well as in the listing file.

> **Note:** After any of these responses, before pressing Enter, you can continue the response with a comma and the answer to what would be the next prompt, without having to wait for that prompt. If you end any response with the semicolon (;), the remaining responses are all assumed to be the default. Processing begins immediately with no further prompting.

Option 2 - Command Line

From your keyboard, type:

LINK *objlist,runfile,mapfile,liblist* [*parm*]...;

objlist is a list of object modules separated by plus signs (+) or spaces.

runfile is the name you want to give the run file.

mapfile is the name you want to give the linker map.

liblist is a list of the libraries to be used, separated by plus signs (+) or spaces.

parm is an optional linker parameter. Each parameter must begin with a slash (/).

The linker is loaded and immediately performs the tasks indicated by the command line.

When you use this command line, the prompts described in Option 1 are not displayed if you specified an entry for all four files or if the command line ends with a semicolon.

If an incomplete list is given and no semicolon is used, the linker prompts for the remaining unspecified files.

Each prompt displays its default, which is accepted by pressing the Enter key, or overridden with an explicit file name or device name. However, if an incomplete list is given and the command line is terminated with a final semicolon, the unspecified files default without further prompting. The *parm*s are never prompted for, but may be added to the end of the command line or to any file specification given in response to a prompt.

Certain variations of this command line are permitted.

Examples:

LINK module

The object module is MODULE.OBJ. A prompt is given, showing the default of MODULE.EXE. After the response is entered, a prompt is given showing the default of NUL.MAP. After the response is given, a prompt is displayed showing the default extension of .LIB.

LINK module;

If the semicolon is added, no further prompts are displayed. The object module of MODULE.OBJ is linked, the run file is put into MODULE.EXE, and no list file is produced.

LINK module,,;

This is similar to the preceding example, except the list file is produced in MODULE.MAP.

LINK module,,

Using the same example, but without the semicolon, MODULE.OBJ is linked, and the run file is produced in MODULE.EXE, but a prompt is given with the default of MODULE.MAP.

LINK module,,NUL;

No list file is produced. The run file is in MODULE.EXE. No further prompts are displayed.

Option 3 - Automatic Responses

It is often convenient to save responses to the linker for use at a later time. This is especially useful when long lists of object modules need to be specified.

Before using this option, you must create the automatic response file. It contains several lines of text, each of which is the response to a linker prompt. These responses must be in the same order as the linker prompts that were discussed earlier in this chapter. If desired, a long response to the object module or libraries prompt may be contained on several lines by using a plus sign (+) to continue the same response onto the next line.

To specify an automatic response file, you enter a file specification preceded by an @ symbol in place of a prompt response or part of a prompt response. The prompt is answered by the contents of the diskette file. The file specification cannot be a reserved DOS file name.

From your keyboard, type:

LINK @[*d*:][path]*filename*[*.ext*]

Use of the file name extension is optional and can be any name. There is no default extension.

Use of this option permits the command that starts LINK to be entered from the keyboard or within a batch file without requiring any response from you.

Example

```
Automatic Response File - RESP1

    MODA+MODB+MODC+
    MODD+MODE+MODF

Automatic Response File - RESP2

    runfile/p
    printout

Command line

    LINK @RESP1+mymod,@RESP2;
```

Notes:

1. The plus sign at the end of the first line in RESP1 causes the modules listed in the first two lines to be considered as the input object modules. After reading RESP1, the linker returns to the command line and sees **+mymod**, so it includes MYMOD.OBJ in the first list of object modules as well.

2. Each of the above lines ends when you press the Enter key.

Example Linker Session

This example shows you the type of information that is displayed during a linker session.

When you type:

```
b:link
```

at the DOS prompt, the system responds with the following messages and prompts, which you answer as shown:

```
The IBM Personal Computer Linker
Version 2.40 (C)Copyright International
Business Machines Corp. 1981, 1985
(C)Copyright Microsoft Corp. 1981, 1985

Object Modules [.OBJ]: example
Run File [EXAMPLE.EXE]: /map
List File [NUL.MAP]: prn/line
Libraries [.LIB]:
```

Notes:

1. By specifying **/map**, you get both an alphabetic and a chronological listing of public symbols.

2. By responding **prn** to the list file prompt, you send your output to the printer.

3. By specifying the /LINE parameter, LINK gives a listing of all line numbers for all modules. (The /LINE parameter can generate a large amount of output.)

4. By pressing Enter in response to the libraries prompt, an automatic library search is performed.

Once LINK locates all libraries, the linker map displays a list of segments in the relative order of their appearance within the load module. The list looks like this:

```
Start   Stop    Length  Name            Class

ØØØØØH ØØØ28H ØØ29H   MAINQQ          CODE
ØØØ3ØH ØØØF6H ØØC7H   ENTXQQ          CODE
ØØ1ØØH ØØ1ØØH ØØØØH   INIXQQ          CODE
ØØ1ØØH Ø38D3H 37D4H   FILVQQ_CODE     CODE
Ø38D4H Ø4921H 1Ø4EH   FILUQQ_CODE     CODE
.
.
.
Ø74AØH Ø74AØH ØØØØH   HEAP            MEMORY
Ø74AØH Ø74AØH ØØØØH   MEMORY          MEMORY
Ø74AØH Ø759FH Ø1ØØH   STACK           STACK
Ø75AØH Ø7925H Ø386H   DATA            DATA
Ø793ØH Ø82A9H Ø97AH   CONST           CONST
```

The information in the **Start** and **Stop** columns shows a 20-bit hex address of each segment relative to location zero. Location zero is the beginning of the load module. The addresses displayed are not the absolute addresses of where these segments are loaded. To find the absolute address of a segment, you must determine where the segment listed as being at relative zero is actually loaded; then add the absolute address to the relative address shown in the linker map. The procedure used to determine where relative zero is actually located is discussed in this chapter, in the section called "How to Determine the Absolute Address of a Segment."

Because you specified the /MAP parameter, the public symbols are displayed by name and by value. For example:

```
Address         Publics by Name

Ø492:ØØØ3H     ABSNQQ
Ø6CD:Ø29FH     ABSRQQ
Ø492:ØØA3H     ADDNQQ
Ø6CD:ØØ87H     ADDRQQ
Ø6Ø2:ØØØFH     ALLHQQ
.
.
.
ØØ1Ø:1BCEH     WT4VQQ
ØØ1Ø:1D7EH     WTFVQQ
ØØ1Ø:1887H     WTIVQQ
ØØ1Ø:19E2H     WTNVQQ
ØØ1Ø:11B2H     WTRVQQ

Address         Publics by Value

ØØØØ:ØØØ1H     MAIN
ØØØØ:ØØ1ØH     ENTGQQ
ØØØØ:ØØ1ØH     MAINQQ
ØØØ3:ØØØØH     BEGXQQ
ØØØ3:ØØ95H     ENDXQQ
.
.
.
F82B:F31CH     CRCXQQ
F82B:F31EH     CRDXQQ
F82B:F322H     CESXQQ
F82B:F5B8H     FNSUQQ
F82B:F5EØH     OUTUQQ
```

The addresses of the public symbols are in the *segment:offset* format, showing the location relative to zero as the beginning of the load module. In some cases, an entry may look like this:

```
F8CC:EBE2H
```

This entry appears to be the address of a load module that is almost 1 megabyte in size. Actually, the area being referenced is relative to a segment base that is pointing to a segment below the relative zero beginning of the load module. This condition produces a pointer that has effectively gone negative. The memory map on the previous page illustrates this point.

When LINK has completed, the following message is displayed:

```
Program entry point at ØØØ3:ØØØØ
```

How to Determine the Absolute Address of a Segment

The linker map displays a list of segments in the relative order of their appearance within the load module. The information displayed shows a 20-bit hex address of each segment relative to location zero. The addresses displayed are not the absolute addresses where these segments are located. To determine where relative zero is actually located, you must use DEBUG, which is described in detail in Chapter 13.

Using DEBUG,

1. Load the application. Note the segment value in CS and the offset within that segment to the entry point as shown in IP. The last line of the linker map also describes this entry point, but

uses relative values, not the absolute values shown by CS and IP.

2. Subtract the relative entry as shown at the end of the map listing from the CS:IP value. For example, let's say CS is at 05DC and IP is at zero.

   The linker map shows the entry point at 0100:0000. (0100 is a segment ID or paragraph number; 0000 is the offset into that segment.)

   In this example, relative zero is located at 04DC:0000, which is 04DC0 absolute.

If a program is loaded low, the relative zero location is located at the end of the Program Segment Prefix, in the location DS plus 100H.

Messages

All messages, except for the warning messages, cause the LINK session to end. Therefore, after you locate and correct a problem, you must rerun LINK.

Messages appear both in the list file and on the display unless you direct the list file to CON, in which case the display messages are suppressed.

All of the linker messages are included in Appendix A of the *DOS Reference.*

EXE2BIN Command

Purpose:

Converts .EXE files to .COM or .BIN files.

Format:

[*d:*][*path*]EXE2BIN [*d:*][*path*]*filename*[*.ext*]
[*d:*][*path*][*filename*[*.ext*]]

Type:

Internal External

***

Remarks:

Specify the parameters:

[*d:*][*path*] before EXE2BIN to specify the drive specifier and path that contains the EXE2BIN command file.

[*d:*][*path*][*filename*][*.ext*] to specify the input file. If you *do not* specify:

[*d:*] the default drive is assumed

[*path*] the current directory is assumed

[*.ext*] .EXE is assumed

[*d:*][*path*]*filename*[*.ext*] to specify the output file. If you *do not* specify:

[*d:*] the drive of the input file is assumed

[*path*] the current directory is assumed

filename the input file name is assumed

[*.ext*] .BIN is assumed

The input file is converted to .COM file format (memory image of the program) and placed in the output file.

The input must be in valid .EXE format as produced by the Linker. The *resident*, or actual code and data, part of the file must be less than 64K. There must be no STACK segment.

Two kinds of conversions are possible, depending on the specified initial CS:IP:

- If CS:IP is not specified in the program (the .EXE file contains 0:0), a pure binary conversion is assumed. If segment fixups are necessary (the program contains instructions requiring segment relocation), you are prompted for the fixup value. This value is the absolute segment at which the program is to be loaded.

  In this case, the resultant program is usable only when loaded at the absolute memory address specified by a user application. The DOS command processor will not be capable of properly loading the program.

- If CS:IP is specified as 0000:100H, it is assumed that the file is to be run as a .COM file, with the location pointer set at 100H by the assembler statement ORG. No segment fixups are allowed, as .COM files must be segment relocatable. Further information is available in Chapter 7 of this book. Once the conversion is complete, you may rename the resultant file to a .COM extension. Then the command processor is capable of loading and executing the program in the same manner as the .COM programs supplied on your DOS diskette.

If CS:IP does not meet one of these criteria, or if it meets the .COM file criterion but has segment fixups, the following message is displayed:

```
File cannot be converted
```

This message is also displayed if the file is not a valid .EXE file.

The following piece of assembler code will cause a segment fixup because the SEG operator is used (either explicitly or implicitly).

```
symbol1  db  "e:\filename",0
symbol2  db  SEG symbol1 ;explicit use of SEG
symbol3  db  symbol1     ;implicit use of SEG
```

The following piece of assembler code will cause a segment fixup because a segment name is used as an immediate field of an instruction.

```
myseg   SEGMENT  PUBLIC
            .
            .
            .
            .
MOV   data1,mseg          ;causes fixup
```

To produce standard .COM files with the assembler, you must both use the assembler statement ORG to set the location pointer of the file at 100H and specify the first location as the start address. (This is done in the END statement.) Also, the program must not use references that are defined only in other segments. For example, with the IBM Personal Computer MACRO Assembler:

```
ORG 1ØØH
START:
.
.
.
END START
```

EXE2BIN resides on your DOS 3.30 Utilities diskette.

Chapter 13. DEBUG Program

Introduction

This chapter explains how to use the DEBUG program.

The DEBUG program can be used to:

- Provide a controlled testing environment so you can monitor and control the execution of a program to be debugged. You can fix problems in a program directly, and then execute the program immediately to determine if the problems have been resolved. You do not need to reassemble a program to find out if your changes worked.
- Load, alter, or display any file.
- Execute *object files*. Object files are executable programs in machine language format.

How to Start the DEBUG Program

To start DEBUG, type:

DEBUG [*d:*][*path*][*filename*[*.ext*]][*parm1*][*parm2*]

If you enter *filename*, the DEBUG program loads the specified file into memory. You may now type commands to alter, display, or execute the contents of the specified file.

If you do *not* enter a file name, you must either work with the present memory contents, or load the required file into memory by using the Name and Load commands. Then you can type commands to alter, display, or execute the memory contents.

The optional parameters, *parm1* and *parm2*, represent the optional parameters for the named *filespec*. For example,

```
DEBUG DISKCOMP.COM A: B:
```

In this command, the A: and B: are the parameters that DEBUG prepares for the DISKCOMP program.

When the DEBUG program starts, the registers and flags are set to the following values for the program being debugged:

- The segment registers (CS, DS, ES, and SS) are set to the bottom of free memory; that is, the first segment after the end of the DEBUG program.
- The Instruction Pointer (IP) is set to hex 0100.
- The Stack Pointer (SP) is set to the end of the segment, or the bottom of the transient portion of the program loader, whichever is lower. The segment size at offset 6 is reduced by hex 100 to allow for a stack that size.

- The remaining registers (AX, BX, CX, DX, BP, SI, and DI) are set to zero. However, if you start the DEBUG program with a filespec, the CX register contains the length of the file in bytes. If the file is greater than 64K, the length is contained in registers BX and CX (the high portion in BX).

- The initial state of the flags is:

```
NV UP EI PL NZ NA PO NC
```

- The default disk transfer address is set to hex 80 in the code segment.

All of available memory is allocated; therefore, any attempt by the loaded program to allocate memory fails.

Notes:

1. If a file loaded by DEBUG has an extension of .EXE, DEBUG does the necessary relocation and sets the segment registers, stack pointer, and Instruction Pointer to the values defined in the file. The DS and ES registers, however, point to the Program Segment Prefix at the lowest available segment. The BX and CX registers contain the size of the program (smaller than the file size).

   The program is loaded at the high end of memory if the appropriate parameter was specified when the linker created the file. Refer to ".EXE File Structure and Loading" in Chapter 10 of this book for more information about loading .EXE files.

2. If a file loaded by DEBUG has an extension of .HEX, the file is assumed to be in INTEL hex format, and is converted to executable form while being loaded.

The DEBUG Command Parameters

Parameter	Definition
address	Enter a one- or two-part designation in one of the following formats: • An alphabetic segment register designation, plus an offset value, such as: CS:0100 • A segment address, plus an offset value, such as: 4BA:0100 • An offset value only, such as: 100 (In this case, each command uses a default segment.) **Note:** 1. In the first two formats, the colon is required to separate the values.

Parameter	Definition
address	2. All numeric values are *hexadecimal* and may be entered as 1-4 characters. 3. The memory locations specified in address must be valid; that is, they must actually exist. Unpredictable results occur if an attempt is made to access a nonexistent memory location.
byte	Enter a 1 or 2 character *hexadecimal* value.
drive	Enter 1 or 2 digits (for example, 0 for drive A or 1 for drive B) to indicate which drive data is to be loaded from or written to. (Refer to the Load and Write commands.)

Parameter	Definition
filespec	Enter a one- to three-part file specification consisting of a drive designation, file name, and file name extension. All three fields are optional. However, for the Name command to be meaningful, you should at least specify a drive designator or a file name. (Refer to the Name command.)
list	Enter 1 or more byte and/or string values. For example, `F3 "XYZ' 8D 4 "abcd"` has five items in the list (that is, three byte entries and two string entries having a total of 10 bytes).

Parameter	Definition
portaddress	Enter a 1-4 character *hexadecimal* value to specify an 8- or 16-bit port address. (Refer to the Input and Output commands.)
	Enter either of the following formats to specify the lower and upper addresses of a range: • *address address* For example: CS:100 110 **Note:** Only an offset value is allowed in the second address. The addresses must be separated by a space or comma.

Parameter	Definition
range	
range	• *address* L *value* where *value* is the number of bytes in *hexadecimal* to be processed by the command. For example: CS:100 L 11 **Notes:** 1. The limit for *range* is hex 10000, so the sum of *value* and the offset part of *address* cannot exceed 64K bytes. To specify a *value* of 64K bytes within four *hexadecimal* characters, enter 0000 (or 0). 2. The memory locations specified in *range* must be valid; that is, they must actually exist. Unpredictable results will occur if an attempt is made to access a non-existent memory location.

Parameter	Definition
registername	Refer to the Register command.
sector sector	Enter 1-3 character *hexadecimal* values to specify: 1. The starting relative sector number 2. The number of sector numbers to be loaded or written In DEBUG, relative sectors are obtained by counting the sectors on the disk surface. The sector at track 0, sector 1, head 0 (the first sector on the disk) is relative sector 0. The numbering continues for each sector on that track and head, then continues with the first sector on the next head of the same track. When all sectors on all heads of the track have been counted, numbering continues with the first sector on head 0 of the next track. **Note:** This is a change from the sector mapping used by DOS Version 1.10. The maximum number of sectors that can be loaded or written with a single command is hex 80. A sector contains 512 bytes. (Refer to the Load and Write commands.)

Parameter	Definition
string	Enter characters enclosed in quotation marks. The quotation marks can be either single (') or double (" "). The ASCII values of the characters in the string are used as a list of byte values. Within a string, the *opposite* set of quotation marks can be used freely as characters. However, if the *same* set of quotation marks (as the delimiters) must be used within the string, then the quotation marks must be doubled. The doubling does not appear in memory. For example: 1. 'This "literal" is correct' 2. 'This ' 'literal' ' is correct' 3. 'This 'literal' is not correct' 4. 'This "'literal'" is not correct' 5. "This 'literal' is correct" 6. "This "'literal'" is correct" 7. "This "literal" is not correct" 8. "This ' 'literal' ' is not correct"

Parameter	Definition
string	In the second and sixth cases above, the word *literal* is enclosed in one set of quotation marks in memory. In the fourth and eighth cases above, the word *literal* is not correct unless you really want it enclosed in two sets of quotation marks in memory.

Parameter	Definition
value	Enter a 1-4 character *hexadecimal* value to specify: • The numbers to be added and subtracted (refer to the Hexarithmetic command), or • The number of instructions to be executed by the Trace command, or • The number of bytes a command should operate on. (Refer to the Trace, Proceed, and Hexarithmetic commands.)

The DEBUG Commands

This section presents a detailed description of how to use the commands to the DEBUG program. The commands appear in alphabetic order; each with its format and purpose. Examples are provided where appropriate.

Information Common to All DEBUG Commands

The following information applies to the DEBUG commands:

- A command is a single letter, usually followed by one or more parameters.
- Commands and parameters can be entered in uppercase or lowercase, or a combination of both.
- Commands and parameters may be separated by delimiters. Delimiters are only required, however, between two consecutive hexadecimal values. Thus, these commands are equivalent:

```
dcs:1ØØ 11Ø
d cs:1ØØ 11Ø
d,cs:1ØØ,11Ø
```

- Press Ctrl-Break to end commands.
- Commands become effective only after you press the Enter key.
- For commands producing a large amount of output, you can press Ctrl-Num Lock to suspend the display to read it before it scrolls away. Press any other character to restart the display.

- You can use the control keys and the DOS editing keys, described in Chapter 2 of the *DOS User's Guide* while using the DEBUG program.

- If a syntax error is encountered, the line is displayed with the error pointed out as follows:

```
d cs:1ØØ CS:11Ø
           ^error
```

  In this case, the Dump command is expecting the second address to contain only a hexadecimal offset value. It finds the S, which is not a valid hexadecimal character.

- The prompt from the DEBUG program is a hyphen (-).

- The DEBUG program resides on your Technical Reference Utilities diskette.

A (Assemble) Command

Purpose:

To assemble IBM Personal Computer Macro Assembler language statements directly into memory.

Format:

A[*address*]

Remarks:

All numeric input to the Assemble command is in hexadecimal. The assembly statements you enter are assembled into memory at successive locations, starting with the address specified in *address*. If no address is specified, the statements are assembled into the area at CS:0100, if no previous Assemble command was used, or into the location following the last instruction assembled by a previous Assemble command. When all desired statements have been entered, press Enter when you are prompted for the next statement to return to the DEBUG prompt.

DEBUG responds to invalid statements by displaying:

```
^error
```

and redisplaying the current assemble address.

DEBUG supports standard 8086/8088 assembly language syntax (and the 8087 instruction set), with the following rules:

- All numeric values entered are hexadecimal and can be entered as 1-4 characters.
- Prefix mnemonics must be entered in front of the opcode to which they refer. They can also be entered on a separate line.

- The segment override mnemonics are CS:, DS:, ES:, and SS:.

- String manipulation mnemonics must explicitly state the string size. For example, MOVSW must be used to move word strings and MOVSB must be used to move byte strings.

- The mnemonic for the far return is RETF.

- The assembler will automatically assemble short, near, or far jumps and calls depending on byte displacement to the destination address. These can be overridden with the NEAR OR FAR prefix. For example:

```
0100:0500 JMP 502   ;a 2 byte short jump
0100:0502 JMP NEAR 505 ;a 3 byte near jump
0100:0505 JMP FAR 50A  ;a 5 byte far jump
```

  The NEAR prefix can be abbreviated to NE, but the FAR prefix cannot be abbreviated.

- DEBUG cannot tell whether some operands refer to a word memory location or a byte memory location. In this case, the data type must be explicitly stated with the prefix WORD PTR or BYTE PTR. DEBUG will also accept the abbreviations WO and BY. For example:

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

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

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

- Two popular pseudo-instructions have also been included. The DB opcode assembles byte values directly into memory. The DW opcode assembles word values directly into memory. For example:

```
DB    1,2,3,4,"THIS IS AN EXAMPLE"
DB    "THIS IS A QUOTE: "'
DB    "THIS IS A QUOTE:'"

DW    1000,2000,3000:", BACH:"
```

- All forms of the register indirect commands are supported. For example:

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

- All opcode synonyms are supported. For example:

```
LOOPZ    100
LOOPE    100

JA       200
JNBE     200
```

A (Assemble) Command

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

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

Example:

```
C > debug -a200
08B4:0200 xor ax,ax
08B4:0202 mov [bx],ax
08B4:0204 ret
08B4:0205
```

C (Compare) Command

Purpose:

Compares the contents of two blocks of memory.

Format:

C *range address*

Remarks:

The contents of the two blocks of memory are compared; the length of the comparison is determined from the *range*. If unequal bytes are found, their addresses and contents are displayed, in the form:

```
addr1  byte1  byte2  addr2
```

where, the first half (addr1 byte1) refers to the location and contents of the mismatching locations in *range*, and the second half (byte2 addr2) refers to the byte found in *address*.

If you enter only an offset for the beginning address of *range*, the C command assumes the segment contained in the DS register. To specify an ending address for *range*, enter it with only an offset value.

Example:

```
C  100 L20 200
```

The 32 bytes (hex 20) of memory beginning at DS:100 are compared with the 32 bytes beginning at DS:200. L20 is the range.

D (Dump) Command

Purpose:

Displays the contents of a portion of memory.

Format:

D [*address*]

or

D [*range*]

Remarks:

The dump is displayed in two parts:

1. A hexadecimal portion. Each byte is displayed in hexadecimal.

2. An ASCII portion. The bytes are displayed as ASCII characters. Unprintable characters (ASCII ∅ to 31 and 127 to 255) are indicated by a period.

With a 40-column system display format, each line begins on an 8-byte boundary and shows 8 bytes.

With an 80-column system display format, each line begins on a 16-byte boundary and shows 16 bytes. There is a hyphen between the 8th and 9th bytes.

Note: The first line may have fewer than 8 or 16 bytes if the starting address of the dump is not on a boundary. In this case, the second line of the dump begins on a boundary.

D (Dump) Command

The Dump command has two format options:

Option 1

Use this option to display the contents of hex 40 bytes (40-column mode) or hex 80 bytes (80-column mode). For example:

```
D address

  or

D
```

The contents are dumped starting with the specified address.

If you do not specify an address, the D command assumes the starting address is the location following the last location displayed by a previous D command. Thus, it is possible to dump consecutive 40-byte or 80-byte areas by entering consecutive D commands without parameters.

If no previous D command was entered, the location is offset hex 100 into the segment originally initialized in the segment registers by DEBUG.

Note: If you enter only an offset for the starting address, the D command assumes the segment contained in the DS register.

D (Dump) Command

Option 2

Use this option to display the contents of the specified address range. For example:

```
D range
```

Note: If you enter only an offset for the starting address, the D command assumes the segment contained in the DS register. If you specify an ending address, enter it with only an offset value.

For example:

```
D cs:100 10C
```

A 40-column display format might look like this:

```
Ø4BA:Ø1ØØ  42 45 52 54 41 2Ø 54 ØØ
                    BERTA T.

Ø4BA:Ø1Ø8  2Ø 42 4F 52 47
                      BORG
```

Purpose:

The Enter command has two modes of operation:

- Replaces the contents of one or more bytes, starting at the specified address, with the values contained in the list (see Option 1).
- Displays and allows modification of bytes in a sequential manner (see Option 2).

Format:

E *address* [*list*]

Remarks:

If you enter only an offset for the address, the E command assumes the segment contained in the DS register.

The Enter command has two format options:

Option 1

Use this option to place the list in memory beginning at the specified address.

```
E  address list
```

For example:

```
E ds:1ØØ F3 "xyz" 8D
```

Memory locations ds:100 through ds:104 are filled with the 5 bytes specified in the list.

Option 2

Use this option to display the address and the byte of a location, then the system waits for your input.

E (Enter) Command

For example:

```
E  address
```

Enter a 1- or 2-character *hexadecimal* value to replace the contents of the byte; then take any of the next three actions:

1. Press the space bar to advance to the next address. Its contents are displayed. If you want to change the contents take option 1, above.

   To advance to the next byte without changing the current byte, press the space bar again.

2. Enter a hyphen to back up to the preceding address. A new line is displayed with the preceding address and its contents. If you want to change the contents, take option 1, above.

   To back up one more byte without changing the current byte, enter another hyphen.

3. Press the Enter key to end the Enter command.

Note: Display lines can have 4 or 8 bytes of data, depending on whether the system display format is 40- or 80-column. Spacing beyond an 8-byte boundary causes a new display line, with the beginning address, to be started.

For example:

```
E cs:1Ø0
```

might cause this display:

```
Ø4BA:Ø1Ø0 EB._
```

To change the contents of 04BA:0100 from hex EB to hex 41, enter 41.

```
Ø4BA:Ø1Ø0 EB.41_
```

To see the contents of the next three locations, press the space bar three times. The screen might look like this:

```
Ø4BA:Ø1Ø0  EB.41  1Ø. Ø0.  BC._
```

To change the contents of the current location (04BA:0103) from hex BC to hex 42, enter 42.

```
Ø4BA:Ø1Ø0  EB.41  1Ø. Ø0.  BC.42_
```

Now, suppose you want to back up and change the hex 10 to hex 6F. This is what the screen would look like after entering two hyphens and the replacement byte:

```
Ø4BA:Ø1Ø0 EB.41 1Ø.Ø0.  BC.42-
Ø4BA:Ø1Ø2 Ø0.-
Ø4BA:Ø1Ø1 1Ø.6F_
```

Press the Enter key to end the Enter command. You will see the hyphen prompt.

F (Fill) Command

Purpose:

Fills the memory locations in the range with the values in the list.

Format:

F *range list*

Remarks:

If the list contains fewer bytes than the address range, the list is used repeatedly until all the designated memory locations are filled.

If the list contains more bytes than the address range, the extra list items are ignored.

Note: If you enter only an offset for the starting address of the range, the Fill command assumes the segment contained in the DS register.

Example:

```
F 4BA:100 L 5 F3 "XYZ" 8D
```

Memory locations 04BA:100 through 04BA:104 are filled with the 5 bytes specified. Remember that the ASCII values of the list characters are stored. Thus, locations 100-104 will contain F3 58 59 5A 8D.

Purpose:

Executes the program you are debugging.

Stops the execution when the instruction at a specified address is reached (breakpoint), and displays the registers, flags, and the next instruction to be executed.

Format:

G [=*address*] [*address* [*address*...]]

Remarks:

Program execution begins with the current instruction, whose address is determined by the contents of the CS and IP registers, unless overridden by the =*address* parameter (the = must be entered). If =*address* is specified, program execution begins with CS:=*address*.

The Go command has two format options:

Option 1

Use this option to execute the program you are debugging without breakpoints. For example:

```
G [=address]
```

This option is useful when testing program execution with different parameters each time. (Refer to the Name command.) Be certain the CS:IP values are set properly before issuing the G command, if not using =*address*.

G (Go) Command

Option 2

This option performs the same function as Option 1 but, in addition, allows breakpoints to be set at the specified addresses. For example:

```
G [=address] address
  [address...]
```

This method causes execution to stop at a specified location so the system/program environment can be examined.

You can specify up to ten breakpoints in any order. You may wish to take advantage of this if your program has many paths, and you want to stop the execution no matter which path the program takes.

The DEBUG program replaces the instruction codes at the breakpoint addresses with an interrupt code (hex CC). If *any one* breakpoint is reached during execution, the execution is stopped, the registers and flags are displayed, and all the breakpoint addresses are restored to their original instruction codes. If no breakpoint is reached, the instructions are *not* restored.

Notes:

1. Once a program has reached completion (DEBUG has displayed the "Program terminated normally" message), it is necessary to reload the program before it can be executed again.

2. Make sure that the address parameters refer to locations that contain valid 8088 instruction codes. If you specify an address that does not contain the first byte valid instruction, unpredictable results occur.

3. The stack pointer must be valid and have 6 bytes available for the Go command; otherwise, unpredictable results occur.

4. If only an offset is entered for a breakpoint, the G command assumes the segment contained in the CS register.

5. Do not set breakpoints at instructions in read-only memory (ROM BIOS or ROM BASIC).

   For example:

```
G 1Ø2 1EF 2Ø8
```

Be careful not to set a breakpoint between a segment override indication (such as ES; alone on a line), and the instruction that the override qualifies.

Execution begins with the current instruction, whose address is the current values of CS:IP. The *=address* parameter was not used.

Three breakpoints are specified; assume that the second is reached. Execution stops before the instruction at location CS:1EF is executed, the original instruction codes are restored, all three breakpoints are removed, the display occurs, and the Go command ends.

Refer to the Register command for a description of the display.

H (Hexarithmetic) Command

Purpose:

Adds the two hexadecimal values, then subtracts the second from the first.

Displays the sum and difference on one line.

Format:

H *value value*

Example:

```
H OF 8
17 07
```

The hexadecimal sum of 000F and 0008 is 0017, and their difference is 0007.

Purpose:

Inputs and displays (in hexadecimal) 1 byte from the specified port.

Format:

I *portaddress*

Example:

```
I 2F8
6B
```

The single hexadecimal byte read from port 02F8 is displayed (6B).

L (Load) Command

Purpose:

Loads a file or absolute disk sectors into memory.

Format:

L [*address*[*drive sector sector*]]

Remarks:

The maximum number of sectors that can be loaded with a single Load command is hex 80.

Note: DEBUG displays a message if a disk read error occurs. You can retry the read operation by pressing F3 to re-display the Load command. Then, press the Enter key.

The Load command has two format options:

Option 1

Use this option to load data from the disk specified by *drive* and place the data in memory beginning at the specified *address*. For example:

```
L  address drive sector sector
```

The data is read from the specified starting relative sector (first sector) and continues until the requested number of sectors is read (second sector).

Note: If you only enter an offset for the beginning address, the L command assumes the segment contained in the CS register.

For example, to load data, you might enter:

```
L DS:1ØØ 1 ØF 6D
```

The data is loaded from the diskette in drive B and placed in memory beginning at DS:100. 6DH (109) consecutive sectors of data are transferred, starting with relative sector hex 0F (15) (the 16th sector on the diskette).

Note: Option 1 cannot be used if the drive specified is a network drive.

Option 2

When issued without parameters, or with only the address parameter, use this option to load the file whose filespec is at CS:80. For example:

```
L

   or

L address
```

This condition is met by specifying the filespec when starting the DEBUG program, or by using the Name command.

Note: If DEBUG was started with a filespec and subsequent Name commands were used, you may need to enter a new Name command for the proper filespec before issuing the Load command.

The file is loaded into memory beginning at CS:100 (or the location specified by *address*), and is read from the drive specified in the filespec (or from the default drive, if none was specified). Note that files with extensions of .COM or .EXE are always loaded at CS:100. If you specified an address, it is ignored.

L (Load) Command

The BX and CX registers are set to the number of bytes read; however, if the file being loaded has an extension of .EXE, BX and CX are set to the actual program size. The file may be loaded at the high end of memory. Refer to the notes in "How to Start the DEBUG Program" at the beginning of this chapter for the conditions that are in effect when .EXE or .HEX files are loaded.

For example:

```
DEBUG
-N myprog
-L
-
```

The file named **myprog** is loaded from the default diskette and placed in memory beginning at location CS:0100.

Purpose:

Moves the contents of the memory locations specified by *range* to the locations beginning at the *address* specified.

Format:

M *range address*

Remarks:

Overlapping moves are always performed without loss of data during the transfer. (The source and destination areas share some of the same memory locations.)

The data in the source area remains unchanged unless overwritten by the move.

Notes:

1. If you enter only an offset for the beginning address of the range, the M command assumes the segment contained in the DS register. If you specify an ending address for the range, enter it with only an offset value.

2. If you enter only an offset for the address of the destination area, the M command assumes the segment contained in the DS register.

Example:

```
M CS:1ØØ 11Ø 5ØØ
```

The 17 bytes of data from CS:100 through CS:110 are moved to the area of memory beginning at DS:500.

N (Name) Command

Purpose:

The Name command has two functions:

- Formats file control blocks for the first two filespecs, at CS:5C and CS:6C. (Starting DEBUG with a filespec also formats a file control block at CS:5C.)

  The file control blocks are set up for the use of the Load and Write commands and to supply required file names for the program being debugged.

- All specified filespecs and other parameters are placed exactly as entered, including delimiters, in a parameter save area at CS:81, with CS:80 containing the number of characters entered. Register AX is set to indicate the validity of the drive specifiers entered with the first two filespecs.

Format:

N [d:][path]*filename*[*.ext*]

Remarks:

If you start the DEBUG program without a filespec, you must use the Name command before a file can be loaded with the L command.

Example:

```
DEBUG
-N myprog
-L
-
```

To define filespecs or other parameters required by the program being debugged, enter:

```
DEBUG myprog
-N file1 file2
-
```

In this example, DEBUG loads the file **myprog** at CS:100, and leaves the file control block at CS:5C formatted with the same filespec. Then, the Name command formats file control blocks for *file1* and *file2* at CS:5C and CS:6C, respectively. The file control block for **myprog** is overwritten. The parameter area at CS:81 contains all characters entered after the **N**, including all delimiters, and CS:80 contains the count of those characters (hex 0C).

O (Output) Command

Purpose:

Sends the *byte* to the specified output port.

Format:

O *portaddress byte*

Example:

To send the byte value 4F to output port 2F8, enter:

```
O 2F8 4F
```

Purpose:

Causes the execution of a subroutine call, a loop instruction, an interrupt, or a repeat string instruction to stop at the next instruction.

Format:

P[=address][value]

Remarks:

When at a subroutine call, a loop instruction, an interrupt, or a repeat string instruction, issue the Proceed command to execute the instruction (as an atomic operation), and return control at the next instruction. The Proceed command has the same syntax as the Trace command. Specifying P0, is the same as specifying T0.

Example:

If the following instructions are executed:

```
0100    CALL    1000
0103    JC      2000
.
.
.
1000    XOR     AX,AX
.
.
.
1XXX    RET
```

And CS:IP was pointing to the CALL 1000 instruction, typing **P** causes the execution of the subroutine and returns control to DEBUG at the JC instruction.

Q (Quit) Command

Purpose:

Ends the DEBUG program.

Format:

Q

Remarks:

The file that you are working on in memory is *not* saved by the Quit command. You must use the Write command to save the file.

DEBUG returns to the command processor which then issues the normal command prompt.

Example:

```
-Q
A>
```

Purpose:

The Register command has three functions:

- Displays the hexadecimal contents of a single register with the option of changing those contents.
- Displays the hexadecimal contents of all the registers, plus the alphabetic flag settings, and the next instruction to be executed.
- Displays the eight 2-letter alphabetic flag settings with the option of changing any or all of them.

Format:

R [*registername*]

Remarks:

When the DEBUG program starts, the registers and flags are set to certain values for the program being debugged. (Refer to "How to Start the DEBUG Program" at the beginning of this chapter.)

Display a Single Register

The valid *registernames* are:

```
AX    BP    SS
BX    SI    CS
CX    DI    IP
DX    DS    PC
SP    ES    F
```

Both IP and PC refer to the instruction pointer.

For example, to display the contents of a single register, you might enter:

```
R AX
```

The system might respond with:

```
AX F1E4
:_
```

Now you may take one of two actions:

- Press Enter to leave the contents unchanged.

  or

- Change the contents of the AX register by entering a 1-4 character hexadecimal value, such as hex FFF.

```
AX F1E4
:FFF_
```

Now pressing Enter changes the contents of the AX register to hex 0FFF.

Display All Registers and Flags

To display the contents of all registers and flags (and the next instruction to be executed), type:

```
R
```

The system might respond with:

```
AX=ØEØØ BX=ØØFF CX=ØØØ7 DX=Ø1FF
SP=Ø39D BP=ØØØØ SI=ØØ5C DI=ØØØØ
DS=Ø4BA ES=Ø4BA SS=Ø4BA CS=Ø4BA
IP=Ø11A  NV UP DI NG NZ AC PE NC
Ø4BA:Ø11A  CD21          INT     21
```

The first four lines display the hexadecimal contents of the registers and the eight alphabetic flag settings. The last line indicates the location of the next instruction to be executed, and its hexadecimal and unassembled formats. This is the instruction pointed to by CS:IP.

Note: A system with an 80-column display shows:

1st line - 8 registers

2nd line - 5 registers and 8 flag settings

3rd line - next instruction information

A system with a 40-column display shows:

1st line - 4 registers

2nd line - 4 registers

3rd line - 4 registers

4th line - 1 register and 8 flag settings

5th line - next instruction information

Display All Flags

There are eight flags, each with 2-letter codes to indicate either a *set* condition or a *clear* condition.

The flags appear in displays in the same order as presented in the following table:

R (Register) Command

Alphabetic Flag Settings

Flag Name	Set	Clear
Overflow (yes/no)	OV	NV
Direction (decrement/increment)	DN	UP
Interrupt (enable/disable)	EI	DI
Sign (negative/positive)	NG	PL
Zero (yes/no)	ZR	NZ
Auxiliary carry (yes/no)	AC	NA
Parity (even/odd)	PE	PO
Carry (yes/no)	CY	NC

To display all flags, enter:

```
R F
```

If all the flags are in a *set* condition, the response is:

```
OV DN EI NG ZR AC PE CY - _
```

Now you can take one of two actions:

- Press Enter to leave the settings unchanged.
- Change any or all of the settings.

To change a flag, just enter its opposite code. The opposite codes can be entered in any order with or without intervening spaces. For example, to change the first, third, fifth, and seventh flags, enter:

```
OV DN EI NG ZR AC PE CY - PONZDINV
```

They are entered in reverse order in this example.

Press Enter and the flags are modified as specified, the prompt appears, and you can enter the next command.

If you want to see if the new codes are in effect, enter:

```
R F
```

The response is:

```
NV DN DI NG NZ AC PO CY - _
```

The first, third, fifth, and seventh flags are changed as requested. The second, fourth, sixth, and eighth flags are unchanged.

Note: A single flag can be changed only once per R F command.

S (Search) Command

Purpose:

Searches the *range* for the character(s) in the *list*.

Format:

S *range list*

Remarks:

All matches are indicated by displaying the addresses where matches are found.

A display of the prompt without an address means that no match was found.

Note: If you enter only an offset for the starting address of the range, the S command assumes the segment contained in the DS register.

Example:

If you want to search the range of addresses from CS:100 through CS:110 for hex 41, type:

```
S CS:1ØØ 11Ø 41
```

If two matches are found the response might be:

```
Ø4BA:Ø1Ø4
Ø4BA:Ø1ØD
```

If you want to search the same range of addresses for a match with the 4-byte list (41 "AB" E), enter:

```
S CS:1ØØ L 11 41 "AB" E
```

The starting addresses of all matches are listed. If no match is found, no address is displayed.

Purpose:

Executes one or more instructions starting with the instruction at CS:IP, or at =*address* if it is specified. The = must be entered. One instruction is assumed, but you can specify more than one with *value*. Displays the contents of all registers and flags *after each* instruction executes. For a description of the display format, refer to the Register command.

Format:

T [=*address*][*value*]

Remarks:

The display caused by the Trace command continues until *value* instructions are executed. Therefore, when tracing multiple instructions, remember you can suspend the scrolling at any time by pressing Ctrl-NumLock. Resume scrolling by entering any other character.

Notes:

1. The Trace command disables all hardware interrupts before executing the user instruction, and then reenables the interrupts when the trap interrupt occurs following the execution of the instruction.

2. TRACE should not be used with any steps that change the contents of the 8259 interrupt mask (ports 20 and 21).

3. If you trace an INT3 instruction, the breakpoint is set at the INT3 location.

T (Trace) Command

Example:

```
T
```

If the IP register contains 011A, and that location contains B40E (MOV AH,0EH), this might be displayed:

```
AX=ØEØØ BX=ØØFF CX=ØØØ7 DX=Ø1FF
SP=Ø39D BP=ØØØØ SI=ØØ5C DI=ØØØØ
DS=Ø4BA ES=Ø4BA SS=Ø4BA CS=Ø4BA
IP=Ø11C  NV UP DI NG NZ AC PE NC
Ø4BA:Ø11C  CD21    INT     21
```

This displays the results *after* the instruction at 011A is executed, and indicates the next instruction to be executed is the INT 21 at location 04BA:011C.

```
T 10
```

Sixteen instructions are executed (starting at CS:IP). The contents of all registers and flags are displayed after each instruction. The display stops after the 16th instruction has been executed. Displays may scroll off the screen unless you suspend the display by pressing the Ctrl-NumLock keys.

U (Unassemble) Command

Purpose:

Unassembles instructions (translates the contents of memory into assembler-like statements) and displays their addresses and hexadecimal values, together with assembler-like statements. For example, a display might look like this:

```
Ø4BA:Ø1ØØ  2Ø6472    AND [SI+72],AH
Ø4BA:Ø1Ø3  FC        CLD
Ø4BA:Ø1Ø4  7665      JBE Ø16B
```

Format:

U [*address*]

or

U [*range*]

Remarks:

The number of bytes unassembled depends on your system display format (40 or 80 columns), and which option you use with the Unassemble command.

Notes:

1. In all cases, the number of bytes unassembled and displayed may be slightly more than either the amount requested or the default amount. This happens because the instructions are of variable lengths; therefore, unassembling the last instruction may result in more bytes than expected.

2. Make sure that the address parameters refer to locations containing valid 8088 instruction codes. If you specify an address that does not contain the first byte of a valid instruction, the display will be incorrect.

3. If you enter only an offset for the starting address, the U command assumes the segment contained in the CS register.

The Unassemble command has two format options:

Option 1

Use this option to either unassemble instructions without specifying an address, or to unassemble instructions beginning with a specified address. For example:

```
U

  or

U address
```

Sixteen bytes are unassembled with a 40-column display. Thirty-two bytes are unassembled while in 80-column mode.

Instructions are unassembled beginning with the specified address.

If you do not specify an address, the U command assumes the starting address is the location following the last instruction unassembled by a previous U command. Thus, it is possible to unassemble consecutive locations, producing continuous unassembled displays, by entering consecutive U commands without parameters.

If no previous U command is entered, the location is offset hex 0100 into the segment originally initialized in the segment registers by DEBUG.

Option 2

Use this option to unassemble instructions in a specified address range. For example:

```
U range
```

All instructions in the specified address range are unassembled, regardless of the system display format.

Note: If you specify an ending address, enter it with only an offset value.

For example:

```
U Ø4ba:Ø1ØØ 1Ø8
```

The display response might be:

```
Ø4BA:Ø1ØØ  2Ø6472     AND [SI+72],AH
Ø4BA:Ø1Ø3  FC         CLD
Ø4BA:Ø1Ø4  7665       JBE Ø16B
Ø4BA:Ø1Ø6  2Ø737Ø     AND [BP+DI+7Ø],DH
```

The same display appears if you enter:

```
U Ø4BA:1ØØ L 7

      or

U Ø4BA:1ØØ L 8

      or

U Ø4BA:1ØØ L 9
```

W (Write) Command

Purpose:

Writes the data being debugged to disk.

Format:

W [*address* [*drive sector sector*]]

Remarks:

The maximum number of sectors that can be written with a single Write command is hex 80.

DEBUG displays a message if a disk write error occurs. You can retry the write operation by pressing F3 to redisplay the Write command, then press the Enter key.

The Write command has two format options:

Option 1

Use this option to write data to disk beginning at a specified address. For example:

```
W address drive sector sector
```

The data beginning at the specified address is written to the disk in the indicated drive. The data is written starting at the specified starting relative sector (first sector) and continues until the requested number of sectors are filled (second sector).

Notes:

1. Be extremely careful when you write data to absolute sectors because an erroneous sector specification destroys whatever was on the disk at that location.

2. If only an offset is entered for the beginning address, the W command assumes the segment contained in the CS register.

3. Remember, the starting sector and the sector count are both specified in *hexadecimal.*

4. Option 1 cannot be used if the drive specified is a network drive.

For example:

```
W 1FD 1 1ØØ A
```

The data beginning at CS:01FD is written to the diskette in drive B, starting at relative sector hex 100 (256) and continuing for hex 0A (10) sectors.

Option 2

This option allows you to use the Write command without specifying parameters or only specifying the address parameter. For example:

```
W

  or

W address
```

When issued as shown above, the Write command writes the file (whose filespec is at CS:80) to disk.

This condition is met by specifying the filespec when starting the DEBUG program, or by using the Name command.

Note: If DEBUG was started with a filespec and subsequent Name commands were used, you may need to enter a new Name command for the proper filespec before issuing the Write command.

In addition, the BX and CX registers must be set to the number of bytes to be written. They may have been set properly by the DEBUG or Load commands, but might have been changed by a Go or Trace command. You must be certain the BX and CX registers contain the correct values.

The file beginning at CS:100, or at the location specified by *address*, is written to the diskette in the drive specified in filespec or the default drive if none was specified.

The debugged file is written over the original file that was loaded into memory, or into a new file if the file name in the FCB didn't previously exist.

Note: An error message is issued if you try to write a file with an extension of .EXE or .HEX. These files must be written in a specific format that DEBUG cannot support.

If you find it necessary to modify a file with an extension of .EXE or .HEX, and the exact locations to be modified are known, use the following procedure:

1. RENAME the file to an extension other than .EXE or .HEX.

2. Load the file into memory using the DEBUG or Load command.

3. Modify the file as needed in memory, but do not try to execute it with the Go or Trace commands. Unpredictable results would occur.

4. Write the file back using the Write command.

5. RENAME the file to its correct name.

Appendix A. Using the Library Manager

The Library Manager

The IBM Library Manager allows you to construct and edit object module libraries. Object files and other library files can be added to a library and object modules can be removed and erased from a library.

Command Line Format

The format of the command line is:

LIB [*library-file* [*pagesize*]*operations*
[,[*list-file*]][,[*newlib*]][;]]

library-file is the name of a library file.

pagesize is an optional switch of the form,

```
"/pagesize:N" or "/p:N"
```

where *N* equals:

16, 32, 64, 128, 256, or 512.

By default, libraries under IBM DOS are always multiples of 512 byte blocks. Object modules always start at the beginning of a new block. A block is also called a page. If the size of the object module is less than a block, the rest of the block is filled with null bytes.

When you specify the pagesize in the command line, the library being created or modified contains *N* byte pages.

The size of the library that you are creating or modifying can increase when you specify larger pagesize values. However, the time it takes to link the

library decreases when you use larger pagesizes.

The default value for the pagesize switch is 512 if the library file is being created, or the current pagesize if the library file is being modified.

> **Note:** Version 2.40 of the Linker is included with this version of DOS. Previous versions of the Linker cannot recognize pagesizes less than 512. Therefore, you should always use the latest version of the Linker.

operations is a list of operations to perform. This list contains an operator plus the name of the file you are adding. The default is an empty list; no changes occur. See "Operators," later in this section, for a description of the operators.

list-file is a file name where a cross reference listing will be placed. No default extensions are used.

The default for [,*list-file*] is no list file; a cross-reference is not generated. You are asked for this entry if it is left empty.

newlib defines the name of a library file to be created with the changes specified by the *operations*. The default is the same name as the library file. If you use the default, the original file is renamed to have the extension ".BAK" instead of ".LIB".

The command line can be broken by a carriage return at any point. You are asked for the remaining parts of the command line. If a semicolon ends any field after the library file name, the

remaining fields take on their default values. If you just specify LIB, you are asked for all entries.

Note: You can have a device identification before any of the entries that you specify in the command line.

Operators

The operators recognized by the Library Manager are:

+	Add the contents of an object file or a library file.
−	Erase an object module.
*	Remove an object module into a file whose name is the specified module name plus the extension ".OBJ".

These individual operators can be combined to perform more complex operations. For example:

− +	Replace an object module with the contents of the object file of the same name (plus ".OBJ").
− *	Remove an object module and at the same time erase it.

Many operations may be performed at once. If you want to specify operations on more than one line, follow your last operation with an '&' and a carriage return.

The operations are performed in the following order:

1. Erasures and removals
2. Additions

Erasures and removals are performed in the order in which the specified object modules occur in the library. Additions are performed in the order that you specified.

Examples:

To add the file TEST.OBJ to the library BASIC.LIB without producing a cross-reference, type:

```
LIB BASIC.LIB+TEST.OBJ;
```

Note that the following is the same as the preceding example:

```
LIB BASIC+TEST;
```

Extensions are optional, and they default to .OBJ if omitted. If you are using a library file that is in the operations list, you must specify the .LIB extension.

To erase TEST from BASIC.LIB, type:

```
LIB BASIC-TEST;
```

To replace TEST in the library with a newer version, type:

```
LIB BASIC-+TEST;
```

Note that the following also have the same effect:

```
LIB BASIC-TEST+TEST.OBJ;

LIB BASIC+TEST-TEST;
```

If you want to make the same change, but put the changes in a new library called BASNEW.LIB, any of the following work:

```
LIB BASIC-+TEST,,BASNEW

LIB BASIC-TEST+TEST,,BASNEW

LIB BASIC+TEST-TEST,,BASNEW
```

If you want to create a library of object modules, type:

```
LIB MYSUBS+FILE1.OBJ+FILE2.OBJ+...+FILEN.OBJ.
```

You are asked for the listing file.

Response File

You can place all of your responses to the prompts in a response file. The Library Manager is instructed to read the responses from the response file by the following:

```
LIB @RESP.TXT
```

where RESP.TXT is the file name containing the responses.

Note: RESP.TXT was chosen as an example. Any valid DOS file name can be used.

Cross-Reference Lists

Two types of cross-references are generated:

- Alphabetic list
- Object module list.

The alphabetic list contains the public symbol followed by the object file. For example, the cross-reference for BASCOM20.LIB contains:

```
$SWPA.......SWAP          $SWPB.......SWAP
$SWPC.......SWAP          $SWPD.......SWAP
```

The second list contains the object file "SWAP", the offset value, and the code and data size in hexadecimal.

```
SWAP Offset: 2197ØH Code and data size: 4FH
$SWPA       $SWPB         &SWPC        &SWPD
```

Library Manager Error Messages

Irrecoverable Errors

MESSAGE:	PROBABLE CAUSE:
aborted by user	You did not want to create library
alignment factor too small	Library larger than (pagesize)*64KB
cannot create new library	Directory full or name bad
cannot open indirect file	Response file does not exist
cannot open VM.TMP	Directory full
cannot read from VM	A problem in the Library Manager
cannot rename old library	Bad file name
cannot reopen library	A problem in the Library Manager
cannot write to VM	Diskette full
command syntax error	Typographical error
error writing to cross reference file	Diskette or fixed disk full

MESSAGE:	PROBABLE CAUSE:
error writing to new library	Diskette or fixed disk full
fetch: not allocated	A problem in the Library Manager
free: not allocated	A problem in the Library Manager
insufficient memory	Not enough memory for Library Manager
internal failure	A problem in the Library Manager
invalid file name extension	Extension specified is not .OBJ or .LIB
invalid library	File is not a library file
invalid library name extension	Name does not end in .LIB
invalid object module...	Corrupt object module
mark: not allocated	A problem in the Library Manager
no more virtual memory	Too many public symbols

MESSAGE:	PROBABLE CAUSE:
syntax error	Typographical error
syntax error (bad file spec)	Typo specifying file name
syntax error (bad input)	Illegal character in input
syntax error (switch name expected)	Name not found after '/'
syntax error (switch value expected)	Illegal character after ':'
too many symbols	Too many public symbols
unexpected EOF on command input	Unexpected EOF on response file
unknown switch	Unknown name after '/'
write to extract file failed	Diskette or fixed disk full
write to library file failed	Diskette or fixed disk full

Recoverable Errors.

MESSAGE:	PROBABLE CAUSE:
cannot create extract file...	Directory full
cannot create listing...	Bad file name or directory full
extension illegal—file ignored	Extension is not .LIB or .OBJ
invalid format...; file ignored	Input file is not an object or library
invalid library header	Input file is not a library
module not in library; ignored	Module to extract or delete does not exist
page size too small—ignored	Value of page size switch is less than 16

Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

Q

R

S

T

U

V

W

Z

Numerics

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

80X0945