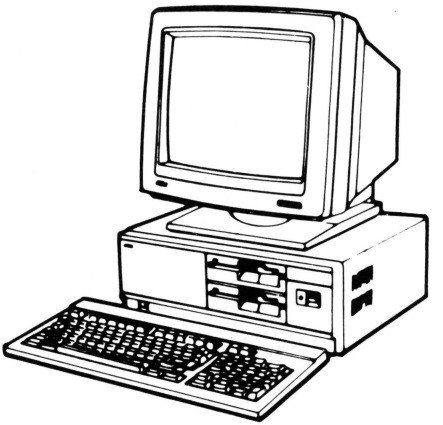
APC III



APC III Supplement to MS™-DOS Programmer's Reference Guide

NEC

NEC Information Systems, Inc.

819-150189-000 Rev. 00

10-84

Important Notice

(1) All rights reserved. This manual is protected by copyright. No part of this manual may be reproduced in any form whatsoever without the written permission of the copyright owner.

(2) The policy of NEC being that of continuous product improvement, the contents of this manual are subject to change, from time to time, without notice.

(3) All efforts have been made to ensure that the contents of this manual are correct; however, should any errors be detected, NEC would greatly appreciate being informed.

(4) NEC can assume no responsibility for errors in this manual or their consequences.

IBM is a registered trademark of International Business Machines Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

MS is a registered trademark of Microsoft Corporation.

CP/M is a registered trademark of Digital Research, Inc.

©Copyright 1984 by NEC Corporation

Contents

Preface ix

Chapter 1 MS-DOS System Files

Overview S1-1
The I/O (Input/Output) System (IO.SYS) S1-2
    Device Drivers S1-2
    ROM BIOS S1-3
The DOS System (MSDOS.SYS) S1-4
The Command Processor (COMMAND.COM) S1-4
    Resident S1-5
    Initialization S1-5
    Transient S1-5
ANSI-Compatible Console Driver (ANSI.SYS) S1-6
Optional Dual RS-232C Serial Port Device Driver (OPTRS232.SYS) S1-8

Chapter 2 MS-DOS Initialization

Initialization Procedures S2-1

Chapter 3 The APC III Video Display Memory (VRAM)

Overview S3-1
VRAM S3-1
    Effects of VRAM on Applications S3-2
    GDCs S3-2
        Master GDC S3-2
        Slave GDC S3-2
    Display Modes S3-3
        Text and Graphic Modes S3-4
        Text Mode VRAM Mapping S3-5

Contents (cont'd)

Chapter 4 The APC III Keyboard Device Drivers

APC III Keyboard S4-1
Keystroke/Shift States Exceptions S4-4
Break Function S4-4
Keyboard System Reboot Function S4-4
Print Screen Image Request S4-5
Alt Entry of Decimal ASCII Code S4-5

Chapter 5 The APC III ROM BIOS

ROM BIOS S5-1
Use of BIOS S5-1
Parameter Passing S5-1

Appendix A MS-DOS Memory Maps SA-1

MS-DOS System Version-Independent Common Memory Area SA-2
CMOS (Battery-Backed Up) Memory Switches SA-8

Appendix B ANSI.SYS Escape Sequences SB-1

ANSI Cursor Control Escape Sequences SB-1
Conventions SB-1
Cursor Functions SB-1
Erasing Functions SB-3
Modes of Operation SB-4
Keyboard Reassignment SB-6

Contents (cont'd)

Appendix C APC III/MS-DOS Interrupt Vector Assignments SC-1

Interrupt Levels SC-1
Extended Bus Interrupt Vectors SC-4
Vectors with Special Meanings SC-5

Appendix D CONFIG.SYS Configuration File SD-1

Configuration Files in MS-DOS V. 2.0 and Higher SD-1
CONFIG.SYS Commands SD-2

Appendix E ROM BIOS Function Calls SE-1

KEYBOARD_IO Call SE-1
CRT_IO Call SE-2
RS232C_IO Call SE-10
PRINTER_IO Call SE-13
TIMER_IO Call SE-14
DISK_IO Call SE-15
OPTRS232C_IO Call SE-17

Appendix F CRT Display Character Set SF-1

Contents (cont'd)

Appendix G VRAM CRT Display SG-1

Text VRAM Memory Map SG-1
Text Mode Attributes SG-2
Color Bits SG-3
Standard Graphics Characters SG-4
Graphics VRAM Memory Map SG-4

Appendix H Keyboard Character Set SH-1

Appendix I APC III Hard Disk Information SI-1

Hard Disk Partition Table SI-1
MS-DOS Partition Format SI-3
System Initialization SI-3
Sample Hard Disk SI-5

Appendix J GRAPH Installable Driver (GRAPHIC.SYS) SJ-1

Applications SJ-1
How to Use the GRAPH Installable Driver SJ-2
Syntax and Data Conventions SJ-2
Operation and Command Descriptions SJ-5
Parameter Syntax Specifications SJ-9
Sample Program SJ-12

Illustrations

Figure	Title	Page
A-1	MS-DOS Memory Map	SA-1
A-2	Parameters for Standard Serial Port (AUX)	SA-9
A-3	Baud Rate for Standard Serial Port (AUX) and First Optional Serial Port (AUX2)	SA-9
A-4	Start of RAMDISK, 8087-2, Printer Data and Interface, and Default Display Color	SA-10
A-5	Parameters for First Optional Serial Port (AUX2)	SA-11
A-6	Printer Data Pins and Port, and RAMDISK	SA-11
A-7	Printer Type and Parallel Printer Parity	SA-12
A-8	Year	SA-12
E-1	Palette Numbers and Codes	SE-5
F-1	The APC III Displayable Character Set	SF-1
H-1	APC III Keyboard Layout	SH-1
H-2	Keyboard Scan Codes/Key Position Conversion Chart	SH-2
J-1	Available Patterns	SJ-11
J-2	Sample M-BASIC Program for Using the GRAPH Device	SJ-12
J-3	Output of M-BASIC Program	SJ-13
J-4	Graphics Display from Output of M-BASIC Program	SJ-14

Tables

Table	Title	Page
3-1	ROM BIOS Display Modes	S3-3
A-1	System Common Memory (I/O Version-Independent)	SA-2
B-1	SGR Parameters	SB-4
B-2	Set Modes	SB-5
C-1	Interrupt Levels	SC-2
C-2	Extended BUS INT Usage	SC-4
G-1	MPU Memory Address Space	SG-1
G-2	Mapping of MPU Memory Addresses	SG-2
G-3	Color Bit Combinations	SG-3
G-4	Palette Registers Input and Output	SG-6
G-5	Modified Palette Registers Input and Output	SG-6
G-6	Available Graphic Modes on the APC III	SG-7
H-1	Keyboard Character Set	SH-3
H-2	Extended Keyboard Character Set	SH-13
H-3	Suggested Keyboard Functions	SH-16
I-1	The Hard Disk Partition Table	SI-2
I-2	Sample Hard Disk	SI-5

Preface

The *APC III MS-DOS Programmer's Reference Guide Supplement* provides information that is specific to MS-DOS on the APC III computer. Information contained in this document is intended primarily for programmers writing software for the APC III using the MS-DOS operating system. The following documents contain additional information on the APC III and the MS-DOS operating system.

MS-DOS User's Guide

APC III Supplement to MS-DOS User's Guide

APC III Owner's Guide

Macro Assembler Manual

*APC III System Reference Guide**

GSX-86 User's Guide

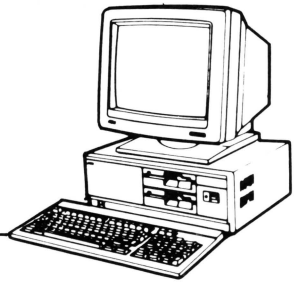
GW-BASIC User's Guide

NEC APC III ROM BIOS listing
(available on diskette)*

NEC μPD7220-GDC Reference Guide

*Available separately

Chapter 1
MS-DOS System Files



Overview

The heart of the MS-DOS Operating System consists of three programs: IO.SYS, MSDOS.SYS, and COMMAND.COM. The first two programs are considered resident; once loaded, they remain in memory until the MS-DOS session is over. The IO.SYS program is the customized part of MS-DOS; it was written by NEC for the APC III. The MSDOS.SYS program provides the basic operating system services. These two programs are stored on the System diskette in hidden files. The file names do not appear in the diskette directory. The COMMAND.COM program is the command processor. It serves as an intermediary between MSDOS.SYS and other programs, for example, applications, utilities, and external MS-DOS commands.

The following three sections describe the IO.SYS, MSDOS.SYS, and COMMAND.COM programs in greater detail. The last section explains the ANSI.SYS program, which provides a set of ANSI-compatible escape sequences for cursor control, screen deletion, display attribute control, and keyboard mapping. Appendix A, MS-DOS Memory Maps, gives additional information on memory maps, independent common memory, and CMOS battery-backup memory.

The I/O (Input/Output) System (IO.SYS)

IO.SYS adapts the APC III hardware-dependent and firmware (ROM BIOS) routines for use with the MS-DOS Operating System.

Device Drivers

The I/O system consists of a set of device driver routines. Each device driver provides access to a particular part of the APC III hardware set. The device drivers can be broken down into those which service "character" devices and those which service "block" devices. IO.SYS presents the DOS (Disk Operating System) with a standard software interface which the DOS can use to request character and block device I/O. This standard interface permits many different hardware sets to use a DOS program.

Character device I/O includes the keyboard, console, printer, and RS-232 serial port. These devices present the I/O system with a serial stream of ASCII characters. Typical I/O system services for character devices include reporting the status of the device, and transferring characters to or from the device as requested by the DOS.

Block device I/O is generally the disk drives, or devices that present the I/O system with a random-access, block-oriented structure. The I/O must provide the DOS with a summary of the device structure, for example, sector size, number of sectors, and allocation unit size. It must also provide the capability to read and write to specific sectors (blocks), since the I/O system does not recognize information as directories or files.

MS-DOS permits another way to incorporate additional device drivers into the I/O system memory other than through the IO.SYS file. These drivers are referred to as "installable" device drivers. When MS-DOS is first loaded, a process called SYSINIT examines the default drive's root directory for the file CONFIG.SYS. The CONFIG.SYS file can contain records specifying additional device drivers to be loaded into memory from disk files before MS-DOS begins to run.

NEC provides an installable driver that is stored in the ANSI.-SYS file. If this driver is loaded at boot time, it becomes the new driver for the console output (CON). When MS-DOS requests output on CON, the new driver from ANSI.SYS replaces the default driver in the IO.SYS file. The ANSI.SYS driver has a special feature. It can implement the ANSI-compatible escape sequences. These character sequences, when sent to the CON device, have special meanings to the device driver. The ANSI.SYS driver will be installed automatically when you load MS-DOS from the system diskette. If you do not wish to have ANSI.SYS loaded, you must modify CONFIG.SYS by removing the DEVICE=ANSI.SYS command. Refer to Appendix B in this supplement for further information.

ROM BIOS

The APC III contains a lower level of software used by the I/O system, called ROM (read-only memory) BIOS (Basic Input/Output System). The ROM BIOS is stored in nonerasable memory chips in the APC III system unit, instead of on diskette. It provides the most primitive level of I/O support for the APC III hardware set. By using ROM BIOS, the IO.SYS file can be kept smaller.

For programmers, there is a major difference between the I/O system and the ROM BIOS. The interface between the DOS and I/O system is hidden to the programmer and therefore cannot be used directly by user programs. The interface between the I/O system and ROM BIOS can be used directly by user programs and can bypass the DOS/IO system. Therefore, access to the hardware is accomplished faster and more directly. Also, certain functions not available through the DOS calls may be accessible via the ROM BIOS calls. Using ROM BIOS will make the software machine dependent on the APC III. For more information, refer to the section "ROM BIOS Interface."

The DOS System (MSDOS.SYS)

The services provided by the I/O system are used by the DOS, which in turn provides basic operating system services, such as program loading, the file and character I/O to COMMAND.-COM and user applications.

The DOS arranges the block devices into logical volumes (diskettes), which are divided into directories, subdirectories, and files. The DOS also provides device-independent file I/O; that is, the ability to open a device as a file, read from it, and write to it, as if it were a file on a block device.

The command processor, user programs, applications, utilities, and external MS-DOS commands make calls to the DOS for services. The calls are made using the 8086 software interrupt instruction with interrupt types 20H-26H, where H indicates hexadecimal. The *MS-DOS Programmer's Reference Guide* contains a detailed description of the available DOS calls. Programmers using languages other than assembler may not need to use these calls explicitly. This is because most language packages that run under MS-DOS translate language I/O statements into DOS calls, transparently to the user. All calls described may be accessed from any high-level language that supports an interface to assembly language subroutines. Refer to Appendix C of this supplement for more information.

The Command Processor (COMMAND.COM)

The command processor is stored on diskette in a file called COMMAND.COM. It is loaded during system boot and comprises three portions:

- resident
- initialization
- transient.

Resident

The resident portion is the part of the command processor that cannot be overwritten. It is loaded into memory immediately following the end of the DOS code in low memory, and remains in memory at all times. All standard MS-DOS error handling is done within the resident portion. This includes displaying error messages and interpreting replies to messages with "Abort, Retry, or Ignore?" prompts. The resident contains routines to process interrupt types 22H (terminate address), 23H (control C exit address), and 24H (fatal error abort address). It also contains the routine needed to reload the transient portion of the command processor.

Initialization

The initialization portion contains code which is executed only once at system startup. It contains the routines necessary to initiate execution of the AUTOEXEC.BAT file (if present). The initialization portion of the command processor is overwritten by the first program that the command processor loads.

Transient

The transient portion contains routines that execute the internal MS-DOS commands, the batch file processor, and the routines that display the "A>" prompt and receive user keyboard commands. The transient portion resides in the high end of memory. It may be overwritten if an application or utility needs the memory that it occupies.

Once the transient portion is overwritten, the resident portion will detect it (using checksum), and attempt to reload it from the COMMAND.COM file using the current default drive. If the volume mounted in the default drive does not contain the COMMAND.COM file, the user will be instructed to insert a diskette with the file. If the default drive is a nonremovable media type, the system will hang. It will be waiting for the user to insert a new diskette in a fixed media drive. To avoid this, it is recommended that COMMAND.COM be copied to the root directory of each nonremovable volume. Alternatively, the SET command may be used to specify COMSPEC, the pathname used in locating COMMAND.COM. Refer to the *MS-DOS User's Guide* for a description of the SET command.

ANSI-Compatible Console Driver (ANSI.SYS)

The ANSI.SYS driver provides a set of ANSI-compatible escape sequences which may be used for cursor control, screen deletion operations, display attribute control, and keyboard remapping.

Before you can access the ANSI.SYS driver, it must be installed. To do this, the files CONFIG.SYS and ANSI.SYS must be present on the boot volume. The CONFIG.SYS file must contain a line reading "DEVICE=ANSI.SYS". This line will direct the SYSINIT module to load the driver from the ANSI.SYS file. These files are provided on the MS-DOS system diskette.

Once the driver is installed as the CON device output driver, any of the ANSI sequences may be sent by calling the character I/O functions of MS-DOS. Characters written to a file named CON, using the file oriented DOS calls, will also be processed by the ANSI driver.

Programs that are written in languages other than assembler may already use these functions via I/O support in the language. For example, in MS-BASIC the "escape [2J" function could be sent to the driver with the following statement:

```
PRINT CHR$(27); "[2J"
```

When executed, the interpreter translates the BASIC PRINT statement into the appropriate calls to the DOS functions needed to display the string. Since the string is a valid ANSI escape sequence (Clear Entire Screen), the driver will clear the screen.

A key remapping function is provided in ANSI.SYS. This escape sequence allows you to change the codes produced by any key (and SHIFT state). Therefore, when keys are pressed, it appears to MS-DOS as if the character(s) programmed for that key have been pressed, instead of the actual characters displayed on the keys.

The maximum length of a string remapped "into" a key that does not use an extended ASCII code is 253 characters. For keys that use an extended ASCII code, the length is 252 characters. There is an overall limit on the number of keys that can be reprogrammed and the total length of the string that the keys are programmed to produce. The limiting factor is the size of the table in the driver that holds the remapping information. The table is 512 bytes long. When the table becomes full, all further requests for remapping are ignored. If a remapping request is made that would require more bytes in the table than are available, the request for further redefinition is ignored.

The amount of table space occupied by a remapping request for expansion to N characters is

Length = 2 + N bytes (for requests to remap keys with standard ASCII codes).

Length = 3 + N bytes (for request to remap keys with extended ASCII codes).

The definitions used by the MS-DOS command-line template-editing feature are present by default, and use approximately 60 of the total bytes in the table.

The KEY utility may be used to determine what remappings are currently in effect, and how much space remains for further definitions. See the *APC III MS-DOS User's Guide Supplement* for information on the KEY utility. Refer to Appendix B for a complete list of the escape sequences implemented by the ANSI.SYS device driver.

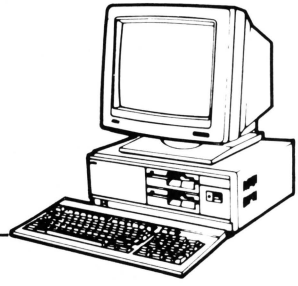
Optional Dual RS-232C Serial Port Device Driver (OPTRS232.SYS)

The OPTRS232.SYS installable device driver provides support for two additional RS-232C serial ports; these two ports are referred to as AUX2 and AUX3. When the driver is installed during system initialization, these two serial ports can be used in the same manner as the standard serial port, AUX. To install the driver, the CONFIG.SYS file must contain a line reading "DEVICE = OPTRS232.SYS". The CONFIG.SYS and OPTRS232.SYS files must be located in the root directory of the system boot drive along with the system files. Remember to copy these files along with the system files when changing or creating a new boot drive.

OPTRS232.SYS also provides a set of routines which function exactly like the ROM BIOS calls to RS232C_IO. These function calls, which are described in Appendix E, can be accessed through interrupt 1EH, OPTRS232C_IO.

The two ports on the APC-H156 option board can be configured with the same parameters that are available for the standard port. The SETCOM utility may be used to configure all of the serial ports. See the section on Custom Utilities in the *APC III Supplement to the MS-DOS User's Guide* for additional information.

Chapter 2
MS-DOS Initialization



Initialization Procedures

The bootstrap loading process is initiated in the following three circumstances:

- when the system is powered on
- the RESET button is pressed after the system is powered on
- the <Ctrl>, <Fnc>, and <Break Stop> keys are pressed simultaneously after the system is powered on.

When the bootstrap-loading process is initiated, control is passed (by hardware) to the initialization code in ROM BIOS. The initialization code then executes several self-diagnostic routines that initialize and check certain aspects of the system hardware such as the 8259 interrupt controller and 7220 GDC (Graphics Display Controller). The system also performs a quick memory test, if the system was just powered on, or the RESET button was pressed.

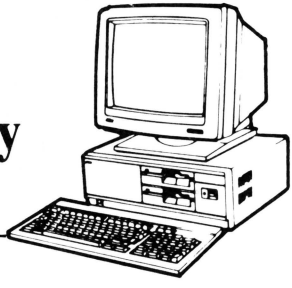
Next, the ROM loader routine searches the block devices for a diskette or hard disk that holds the bootstrap loader in its reserved sector(s). When the unit is found, the bootstrap loader is read into memory from the diskette or disk and control is transferred to the bootstrap loader.

The function of the bootstrap loader is to read the files IO.SYS and MSDOS.SYS into memory from their locations at the start of the data area of the disk or diskette. Control is then passed to an initialization routine in the IO.SYS code. The IO.SYS initialization routine initializes the IO system drivers and then passes control to SYSINIT, which is linked into the IO.SYS file.

SYSINIT searches the diskette or disk for the CONFIG.SYS file. Once CONFIG.SYS is found, SYSINIT reads the file and processes the commands. Refer to Appendix D for details on the CONFIG.SYS commands. If CONFIG.SYS contains "DEVICE =" commands, then the specified installable drivers are loaded into memory (as an extension to the I/O system) from the indicated files.

Finally, SYSINIT reinitializes the DOS code. Control is then passed via an EXEC call to the command processor. When COMMAND.COM initializes, it searches for a file named AUTOEXEC.BAT on the disk unit that the system was loaded with. If this file is found, the prompt for time and date is skipped, and the commands in AUTOEXEC.BAT are executed. By creating an AUTOEXEC.BAT file, you can customize the startup procedures for an APC III. Refer to the *MS-DOS User's Guide* for information on using .BAT (Batch) command files.

Chapter 3
The APC III Video Display Memory (VRAM)



Overview

This section provides the experienced programmer with enough information to directly access the VRAM memory area in conjunction with the ROM BIOS CRT IO function calls. It is not a comprehensive description of the APC III CRT display hardware and will not explain the operation of the NEC 7220 GDC chips. Instead, refer to the *APC III System Reference Guide* and the *NEC μPD7220-GDC Reference Guide.* Appendix E gives detailed information on the ROM BIOS Call Functions.

VRAM

The APC III CRT display is memory mapped. The memory used to hold the data that is displayed (either ASCII characters or graphics) is directly accessible to the 8086 MPU (main processing unit) as an area of memory. This area is called video display memory, or VRAM. VRAM allows exceptionally fast character or graphics display operations, by simply moving data into and out of the VRAM area. Both graphics and text modes of operation are available.

Effects of VRAM on Applications

The MS-DOS I/O system and ROM BIOS contain routines which may be used to send data to the display. Some applications may benefit from the increased performance available by bypassing the DOS and ROM BIOS calls and writing directly to the VRAM. This will also require a little extra work from the programmer because the program will need to keep track of cursor position, scrolling, and the conversions between screen positions and memory addresses within the VRAM.

GDCs

Two NEC 7220 GDCs are at the heart of the CRT display controller hardware. They share access to the display memory areas with the 8086 MPU. The 7220s transform the contents of the VRAM areas into composite video signals required to drive monochrome or color monitors, or RF (Radio Frequency) converter/TV adapters for TV displays.

The first GDC, referred to as the "master," controls the standard APC III text display modes. The second, or "slave," GDC provides standard APC III graphics modes. Each GDC controls its own VRAM memory area.

MASTER GDC

The standard APC III comes with 8 K bytes of text VRAM for the master GDC. The area occupies 16 K bytes of MPU address space; however, only the lower 8 bits (even byte addresses) of each word are actually memory locations. This VRAM area provides enough memory for two text display pages, using eight different colors for each character. Several other character attributes, such as reverse video and underlining, are also available.

SLAVE GDC

The standard APC III is equipped with 64 KB of VRAM for the slave GDC. This provides enough memory for two pages of a single graphics plane of up to 640 × 400 pixels. A color graphics expansion option may be added to provide an additional 128 KB, and permitting the use of three color planes of up to 640 × 400 pixels each.

Display Modes

Any of the modes listed in Table 3-1 may be selected using the ROM BIOS CRT I/O function calls.

Table 3-1 ROM BIOS Display Modes

Mode	Controller	Pages
Text 40 × 25 characters - monochrome	Master GDC	2
Text 40 × 25 characters - color	Master GDC	2
Text 80 × 25 characters - monochrome	Master GDC	2
Text 80 × 25 characters - color	Master GDC	2
Graphics 320 × 200 - monochrome	Slave GDC	2
Graphics 320 × 200 - color	Slave GDC	2
Graphics 640 × 200 - monochrome	Slave GDC	2
Graphics 640 × 200 - color	Slave GDC	2
Graphics 640 × 400 - monochrome	Slave GDC	2
Graphics 640 × 400 - color	Slave GDC	2

Note All color graphic modes require the presence of the Color Graphics option. All other modes are supported on the standard unit.

Any of these modes may be selected using the ROM BIOS CRT I/O function calls (Set A Display Mode). The active display page can also be set with a ROM BIOS call (Select Active Display Page). The hardware allows page selection through a bank-switching arrangement. Independent control of the active page to be displayed on the screen and the active page to be accessible by the MPU (in the address space) is provided by special hardware registers. The ROM BIOS calls always switch both of these features together. For information on direct access of the graphics page select hardware, refer to the *APC III System Reference Guide*.

TEXT AND GRAPHIC MODES

The text modes controlled by the master GDC use a ROM character generator to produce the character set font. Refer to Appendix F for a listing of the displayable characters. In these modes each byte in the VRAM contains the ASCII code of the character to be displayed. Each character has a corresponding byte in the attribute memory area. The ROM character generator is used during the display of a character to convert the ASCII codes into the correct pixels of each scan line that makes up the character. The 40 × 25 character mode must be used for TV displays. CRT monitor type displays can use either 40 × 25 or 80 × 25 character modes.

The ROM BIOS function calls provide many useful services when using these modes, such as cursor positioning, cursor type selection, selection of active page, window scrolling, and character and attribute display.

The graphic modes controlled by the slave GDC use the graphics VRAM as a bit-mapped graphics area. In these modes, each bit in the VRAM corresponds to one pixel on the display.

To produce characters on the screen in the 320 × 200 or 640 × 200 modes, several bytes of data must be stored in the VRAM to activate the necessary bits to form a displayed character. The ROM BIOS CRT write character I/O functions converts the ASCII code into the proper pixel data in VRAM to display the character on the screen. For programs that do not use ROM BIOS, the operation can also be simplified using the capabilities of the 7220 GDC; however the details are not discussed here. A more complex user program could bypass this ROM BIOS feature, and write the necessary data directly to the VRAM. Alternate character fonts can be implemented in this manner.

The APC III hardware is capable of operating both the master and slave 7220 GDCs concurrently. This hardware feature is not. however, supported by MS-DOS or the ROM BIOS calls. When using the ROM BIOS, only a single text or graphics mode is in effect at any time.

In all graphics modes, ROM BIOS offers routines which control individual pixels, thereby freeing the user program for necessary address calculations in converting a logical (x,y) coordinate into a physical VRAM address. Additionally, ROM BIOS calls providing simple graphics figure drawing, including lines, circles, and rectangles, are supplied.

ROM BIOS also provides functions to manipulate the palette registers. This feature allows arbitrary mapping of color plane bit combinations for a single pixel into arbitrary combination of output colors. Access to the registers is provided by the ROM BIOS. In all graphics display modes supported by the ROM BIOS, there is also a mapping switch in hardware (software programmable), allowing the GDC and MPU to bank-switch between two pages of graphics. This operation is supported by the Select Active Display Page function within the CRT_IO ROM BIOS.

The method used to access the VRAM memory will vary from language to language. In assembler, locations within the VRAM can be addressed using the usual segment:offset addressing scheme. From BASIC, a DEFSEG with PEEKs and POKE can be used. In some instances, there may be no way to access the VRAM directly. Instead, simple assembly language subroutines, called from the high-level language, can be written to access arbitrary memory locations.

TEXT MODE VRAM MAPPING

The text mode VRAM mapping provides separate areas for character codes and display attributes. The attribute area may be thought of as overlaying the text area. For example, to cause a reverse video "A" character to be displayed in the first character position, the program would place the ASCII code (41H) in the first character position of the VRAM, and the attribute control byte for reverse-video (04H) in the first character position of the attribute area.

In 80 × 25 text modes, the character codes are stored with the upper left character of the display mapped from the lowest address in the VRAM display page. All subsequent even 8086 addresses contain the characters for the first line. After the 80th character, the next even address is mapped to the first character of the second line of the display, and so forth.

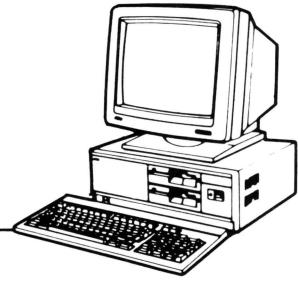
The same arrangement is used in mapping the character attribute areas, so that the attribute byte for a given VRAM location is always 2000H bytes beyond the given address. The same arrangement is also true for 40 × 25 character text modes; however, only every other location from the 80 × 25 VRAM map is used.

The attribute byte corresponding to each character in text VRAM allows the character to be modified in order to display any combinations of several variations of the character. The following attributes are available.

Green color component
Red color component
Blue color component
} these combine into eight colors (for color displays) or intensity control (monochrome displays)
Vertical strike-through
Underline
Reverse Video
Blinking
Overline

In the graphics modes, displays are built of combinations of individual pixels. Various colors are achieved by use of the palette registers and by combining colors through manipulating the three pixels (in the three color planes) which control the color components (red, green, and blue). Refer to Appendix G for information on the VRAM CRT displays.

Chapter 4
The APC III Keyboard Device Drivers



APC III Keyboard

The APC III keyboard contains a total of 101 keys. Each key is classified as a shift key or character key. The following are shift keys.

<Fnc>
<Ctrl>
<Caps Lock>
<Shift> (2)
<Grph 1>
<Grph 2>
<Alt>

The remainder of the keys are character keys. Each key (shift and character) has an assigned scan code which identifies it uniquely. The one exception is the two <Shift> keys, which share a scan code of 45.

Each time a key is pressed or released, an internal 8048 microprocessor, housed in the keyboard unit, sends an 8-bit code in serial form to the APC III system unit. This code contains a 7-bit scan code, which identifies the key, and 1-bit that indicates whether key contact has been made (switch closed) or broken (switch open).

When one of these characters is received, an interrupt occurs to the 8086 MPU. The interrupt activates a routine in ROM BIOS which receives and decodes the character. Since both shift and character key keystrokes are received in this manner, ROM BIOS must keep track of the current shift state as shift keystrokes occur. As character codes are received, ROM BIOS attempts to decode the ASCII code. Since there are more shift-state/character key combinations than a single 8-bit code can encode, not all are mapped directly into ASCII codes.

Certain shift-state/key code combinations are disregarded entirely by ROM BIOS. Others are mapped into an "extended ASCII" code. The extended ASCII codes provide an additional 256 codes which are used to represent selected character key/shift states. The extended ASCII codes can be thought of as an ASCII 00N (NUL) character, followed by the extended ASCII code. This forces the ASCII NUL code itself to be mapped as an extended code.

In most cases, combinations of shift states are undefined. For example, there is no such character as <Shift>, <Ctrl>, and <PF1>, as far as ROM BIOS is concerned. Only a single shift state applies to any single keystroke if the user is actually typing in more than a single shift state. For example, if both <Grph 1> and <Grph 2> are locked down, then ROM BIOS uses the following precedence list to determine which shift state is in effect.

Highest → <Grph 1>, <Grph 2>, <Alt>, <Ctrl>, <Shift>, <Fnc> ← Lowest

There are exceptions to the "only one shift state" rule. <Grph 1> and <Grph 2> can be used along with <Shift> to produce valid and unique extended ASCII codes. Refer to Appendix H for a complete summary of effective shift states.

Once a keystroke has been classified, it is entered into a FIFO (First In First Out) queue as a 16-bit entry. If the character was mapped into a standard ASCII code, then the FIFO entry consists of the key's scan code (8-bits), plus the ASCII code itself (8-bits). If the keystroke was mapped into extended ASCII, then the extended ASCII code (8-bits) plus the value 00H (8-bits) is entered in the FIFO.

Certain character key/shift states do not map into either an ASCII or extended ASCII code. These keystrokes will be discarded (no FIFO entry will be made), and no further processing for the keystrokes will occur.

In the MS-DOS I/O, requests to read the keyboard result in calls to ROM BIOS. The calls determine whether characters are present in the FIFO (for status calls from the DOS), or when to get the next character in the FIFO (for character read calls from the DOS). Within the I/O keyboard driver routines, a final level of keyboard processing occurs. Before returning characters received from ROM BIOS to the DOS, the console input driver (ANSI.SYS only) checks the character against a remapping list. If the character (standard or extended ASCII) has a corresponding entry in the remapping list, then the entry in the list is used to determine which characters the driver will return as the DOS requests them. This arrangement allows the driver complete flexibility in that every key on the keyboard can be remapped.

It should be noted that the driver handles extended codes returned by the ROM BIOS *only* if the extended code has an entry in the key remapping list. Extended ASCII codes received from ROM BIOS which do not have a corresponding entry in the remapping list will be discarded by the I/O, and not returned to the DOS. Therefore, if it is desirable to have an extended ASCII keystroke/shift combination produce a code (or codes) via the DOS INT 21H character function calls, the mapping from extended ASCII to ASCII must be requested with the Keyboard Remapping escape sequence, or the KEY utility.

After system initialization, only the unshifted PF keys (which use extended ASCII) used for the MS-DOS command line template editing functions are defined in the remapping list. All other PF keys (and shift states) will not return codes until a mapping has been specified.

Keystroke/Shift States Exceptions

Certain keystroke/shift states have special predefined meanings, and fall outside of the rules mentioned in the previous discussion. They are recognized immediately by the interrupt handler for the keyboard in ROM BIOS. The following four keystroke/shift states are the exceptions.

- Break function
- Keyboard system reboot function
- Print screen image request
- Alt entry of decimal ASCII code

The following sections explain each of the four keystroke/shift states.

Break Function

The combination of the <Ctrl> shift key and the <Break Stop> character key will result in the keyboard routine signaling 8086 interrupt level 6H with a software interrupt. Also, the extended ASCII code 00H 00H will be returned via ROM BIOS. MS-DOS will not recognize these codes unless they have been remapped. By default, the Break interrupt vector is set to a null routine, which simply performs an IRET instruction. It is up to the user to change the interrupt vector if a user-supplied Break routine is to be implemented.

Keyboard System Reboot Function

The combination of the <Fnc> and <Ctrl> shift keys, along with the <Break Stop> character key, will cause the APC III to perform a "warm start." This feature may be useful in regaining control of the system if a user program is trapped in an infinite loop.

Print Screen Image Request

The combination of the <Shift> key and <Print> character key will cause the text appearing on the CRT display to be printed to the default system printer. The combination of the <Shift> and <Ctrl> shift keys along with the <Print> character key requests that the CRTDUMP resident extension program print a copy of the graphics image currently displayed on the screen.

Neither of these combinations result in any codes being returned for ROM BIOS read keyboard calls.

Alt Entry of Decimal ASCII Code

A special feature of the APC III keyboard allows any ASCII code to be entered directly by typing its ASCII code in decimal. Press the <Alt> key and then type the three-digit decimal value of the desired character on the numeric keypad digit keys. Release the <Alt> key. If more than three characters are typed, a modulo 256 result is created. These three keys are interpreted as a character code. For example, to enter an ASCII DEL(ete) character (code 127, decimal), press the <Alt> key and then type 1, 2, and 7 digit keys on the numeric keypad. Release the <Alt> key. This entire sequence is equivalent to typing the <Del> key once.

Chapter 5
The APC III ROM BIOS

ROM BIOS

ROM BIOS provides device-level control of major I/O devices in the system unit. An assembly language programmer will have the ability to perform block or character-level I/O operations using ROM BIOS routines. The I/O operations can be performed without concern for operating characteristics and device address. System services such as time of day are also provided.

The goal is to provide an operational interface to the system and relieve the programmer from concern over hardware device characteristics. The BIOS interface insulates the user from the hardware allowing new devices to be added to the system, while retaining the BIOS-level interface to the device. In this manner, hardware modifications and enhancements become transparent to user programs.

Use of BIOS

Access to the BIOS functions is through the 8086 software interrupts. Each BIOS entry point is available through its own interrupt, which can be found in the Interrupt Vector listing in Appendix C. It is important to note that calls to the APC III ROM BIOS are not portable to MS-DOS systems running on computers other than the APC III.

Parameter Passing

All parameters passed to and from the BIOS routines go through the 8086 registers. Refer to Appendix E for a summary of the available ROM BIOS calls. For a more detailed discussion of the ROM BIOS, refer to the *APC III System Reference Guide* and the ROM BIOS lists, which reside on a diskette that must be ordered separately.

Appendix A
MS-DOS Memory Maps

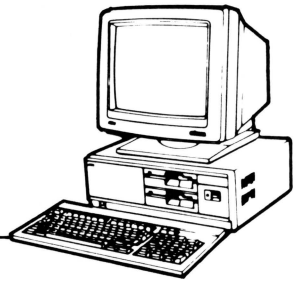


Figure A-1 illustrates an overview of the MS-DOS Memory Map.

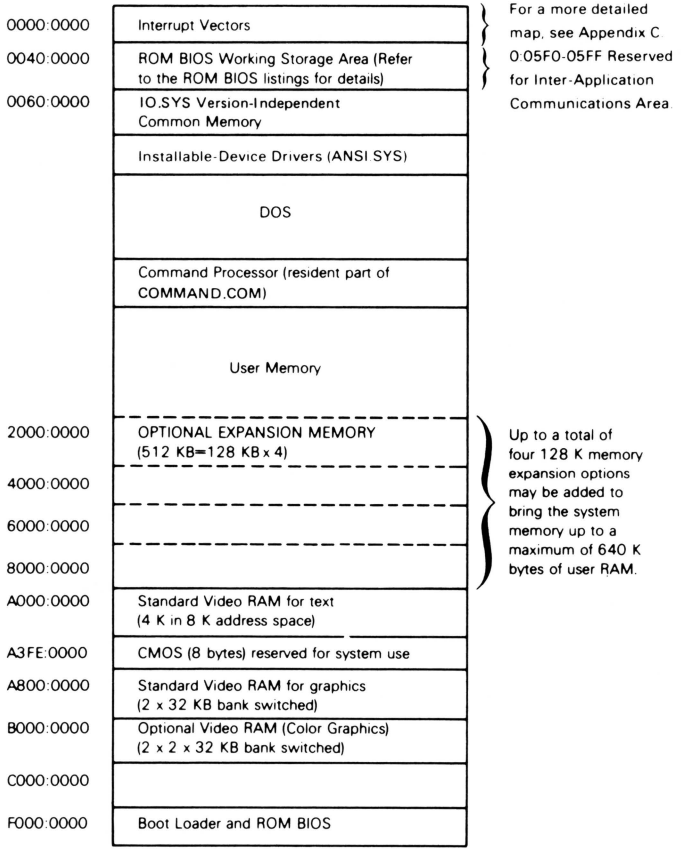


Figure A-1 MS-DOS Memory Map

MS-DOS System Version — Independent Common Memory Area

The System Common Memory area provides access to certain important MS-DOS I/O system items in a version-independent way. The following definitions in Table A-1 apply to all future versions of the MS-DOS I/O for the APC III. Future versions may include additional items not included in this appendix.

Table A-1 System Common Memory (I/O Version Independent)

Address	Length in Bytes	Contents
0060:0000	3	I/O system initialization entry point.
0060:0003	1	I/O system version; ASCII digit, 1,2, etc.
0060:0004	4	Reserved
0060:0008	8	Copy of CMOS contents (see the following section).
0060:0010	4	DWORD pointer to CON device control table.
0060:0014	4	DWORD pointer to AUX device control tables.
0060:0018	4	DWORD pointer to PRN device control table.
0060:001C	4	DWORD pointer to CLK device control table.
0060:0020	4	DWORD pointer to DSK device control table.
0060:0024	8	Reserved
0060:002C	4	DWORD pointer to configuration table.

Table A-i System Common Memory (I/O Version Independent) (cont'd)

CON Device Control Table

Offset	Length in Bytes	Contents
0000	4	DWORD pointer to keyboard remapping control table.
0004	4	Reserved
0008	4	Reserved
000C	4	Reserved

AUX Device Control Tables

Offset	Length in Bytes	Contents
0000	4	AUX receive character buffer pointer.
0004	2	AUX receive character buffer length (bytes).
0006	1	AUX receive time out (seconds).
0007	1	AUX transmit time out (seconds).
0008	1	AUX baud rate (bits 3-0)*
0009	1	AUX parameters*
000A	6	Reserved
0010	4	AUX2 receive character buffer pointer.
0014	2	AUX2 receive character buffer length (bytes).
0016	1	AUX2 receive time out (seconds).

Table A-1 System Common Memory (I/O Version Independent (cont'd)

AUX Device Control Tables (cont'd)

Offset	Length in Bytes	Contents
0017	1	AUX2 transmit time out (seconds).
0018	1	AUX2 baud rate (bits 3-0).*
0019	1	AUX2 parameters*
001A	6	Reserved
0020	4	AUX 3 receive character buffer pointer.
0024	2	AUX3 receive character buffer length (bytes).
0026	1	AUX3 receive time out (seconds).
0027	1	AUX3 transmit time out (seconds).
0028	1	AUX3 baud rate (bits 3-0)*
0029	1	AUX3 parameters*
002A	6	Reserved

* The bit definitions for serial port baud rate and parameters correspond to those defined for CMOS memory switches A3FE:0002 and A3FE:0006.

Note When the driver OPTRS232.SYS for the optional dual port RS-232C board is not installed, the AUX2 and AUX3 receive character buffer pointers will be set to FFFF:FFFF.

Table A-1 System Common Memory (I/O Version Independent (cont'd)

PRN Device Control Table

Offset	Length in Bytes	Contents
0000	1	SO code
0001	1	SI code
0002	3	Reserved
0005	256	Code translation table.

CLK Device Control Table

Offset	Length in Bytes	Contents
0000	4	Reserved for alarm clock control table.
0004	4	Reserved

DSK Device Control Table

Offset	Length in Bytes	Contents
-1	1	Number of physical diskette drives.
-2	1	Number of diskette units defined by driver.
0000	40	Diskette drive 0 control block.
0028	40	Diskette drive 1 control block.
0050	40	Hard disk drive 0 control block.
0078	40	Hard disk drive 1 control block.

Table A-1 System Common Memory (I/O Version Independent (cont'd)

Disk Control Block Format

Offset	Length in Bytes	Contents
0000	1	Current controller sense status.
0001	1	Current I/O command and options.
0002	1	Device address and unit.
0003	10	Parameter block for DISK_IO interrupt.
000D	1	Current return status from DISK_IO.
000E	1	Reserved
000F	1	Reserved
0010	10	Constant value block for current disk.
002A	13	MS-DOS BPB of current disk.
0027	1	Reserved

Parameter Block for DISK_IO Interrupt (Diskette)

Offset	Length in Bytes	Contents
0000	1	Cylinder address
0001	1	Head address
0002	1	Sector address
0003	1	Sector length
0004	2	I/O data byte length
0006	2	I/O data buffer offset
0008	2	I/O data buffer segment

Table A-1 System Common Memory (I/O Version Independent (cont'd)

Parameter Block for DISK_IO Interrupt (Hard Disk)

Offset	Length in Bytes	Contents
0000	4	Sector address
0004	2	I/O data byte length
0006	2	I/O data buffer offset
0008	2	I/O data buffer segment

Constant Value Block for Current Disk (Diskette)

Offset	Length in Bytes	Contents
0000	1	Last cylinder address
0001	1	Last head address
0002	1	Last sector address
0003	7	Reserved

Constant Value Block for Current Disk (Hard Disk)

Offset	Length in Bytes	Contents
0000	4	Start sector address
0004	4	Last sector address
0008	1	Number of partitions
0009	1	Reserved

Table A-1 System Common Memory (I/O Version Independent) (cont'd)

Configuration Table Format

Offset	Length in Bytes	Contents
0000	32	Reserved

CMOS (Battery-Backed Up) Memory Switches

The APC III provides eight bytes of nonvolatile memory for storing certain system operating parameters. Information specific to the particular hardware set can be configured in CMOS once. It will remain in effect until the contents of the CMOS are lost, or until they are changed. All necessary settings of these memory switches can be made via the APC III utilities; however, the maps in Figures A-2 through A-8 are available for programming information.

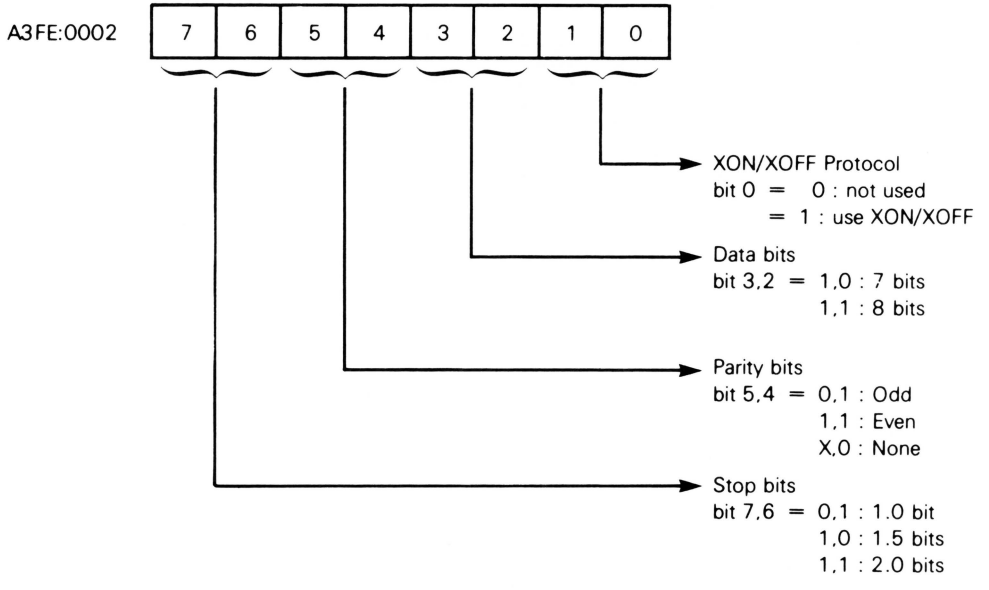


Figure A-2 Parameters for Standard Serial Port (AUX)

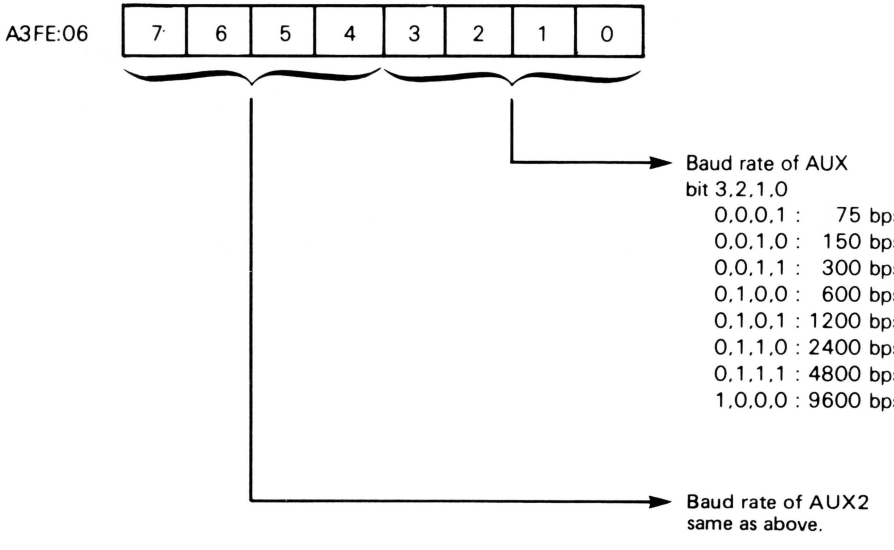


Figure A-3 Baud Rate for Standard Serial Port (AUX) and First Optional Serial Port (AUX2)

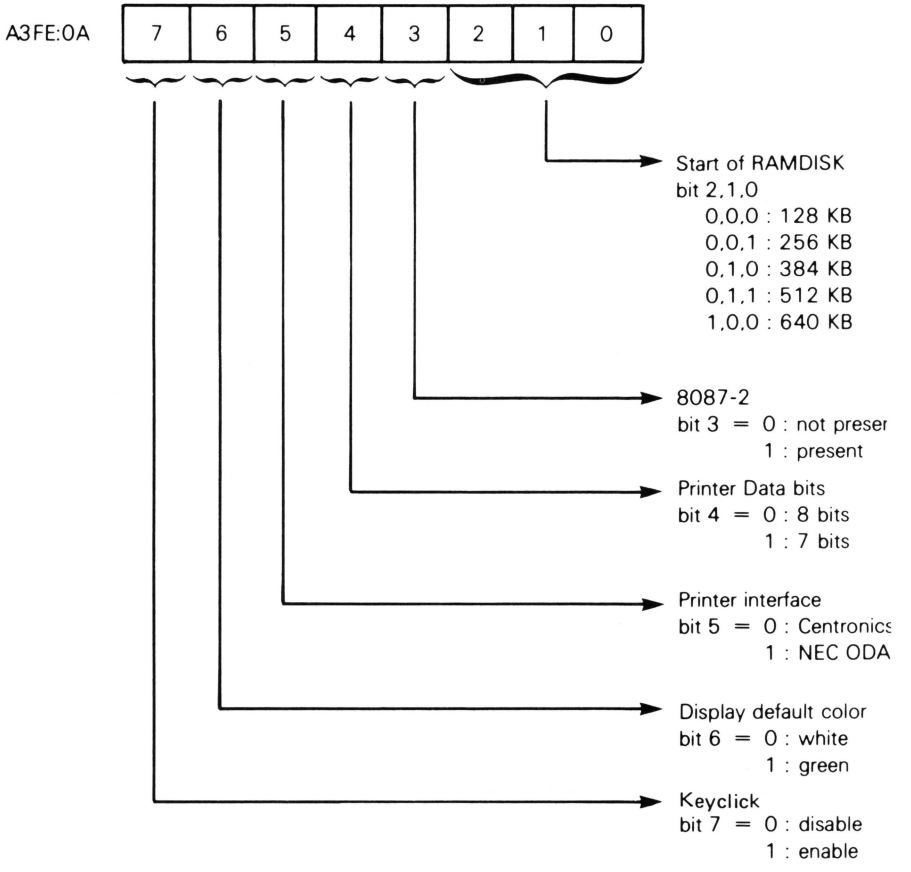


Figure A-4 Start of RAMDISK, 8087-2, Printer Data and Interface, and Default Display Color

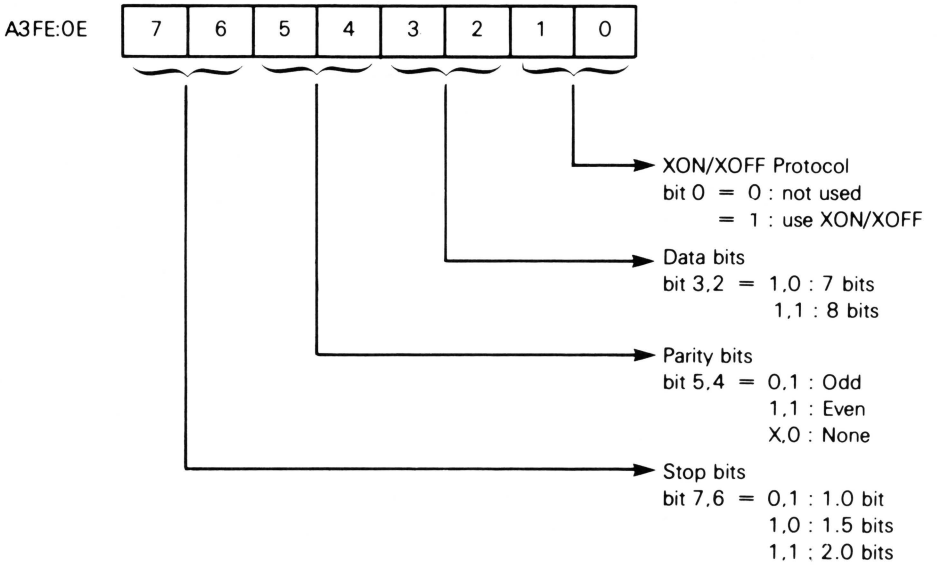


Figure A-5 Parameters for First Optional Serial Port (AUX2)

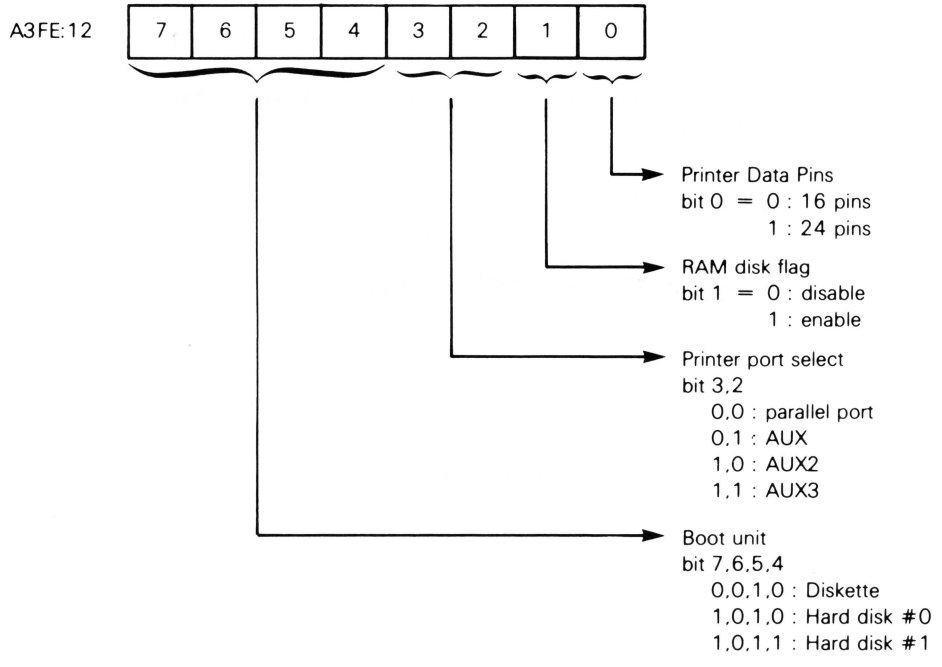


Figure A-6 Printer Data Pins and Port, and RAMDISK

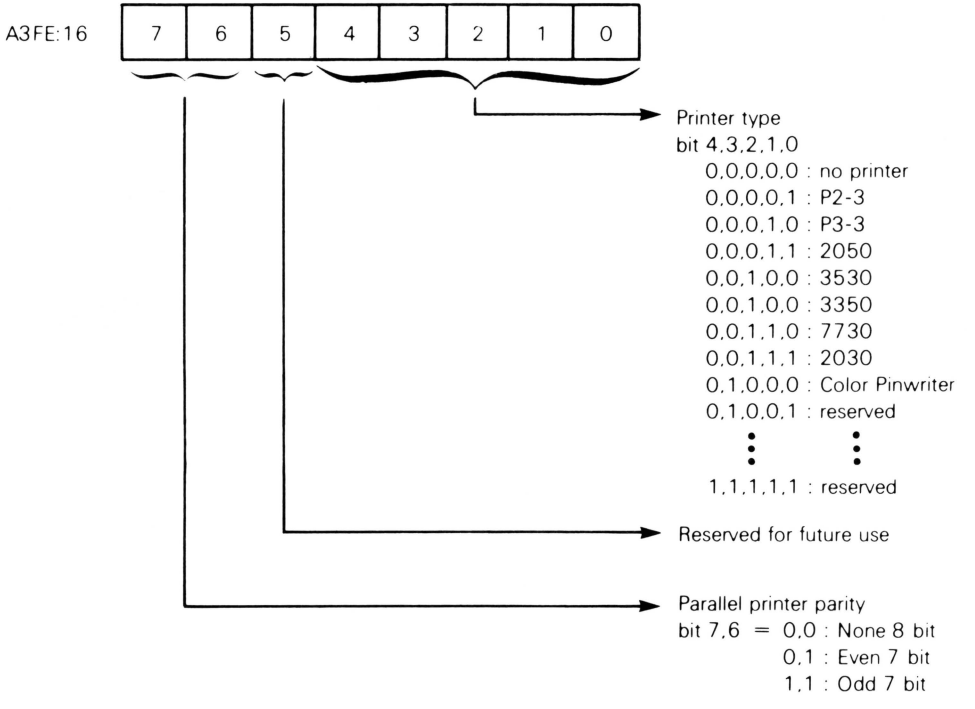


Figure A-7 Printer Type and Parallel Printer Parity

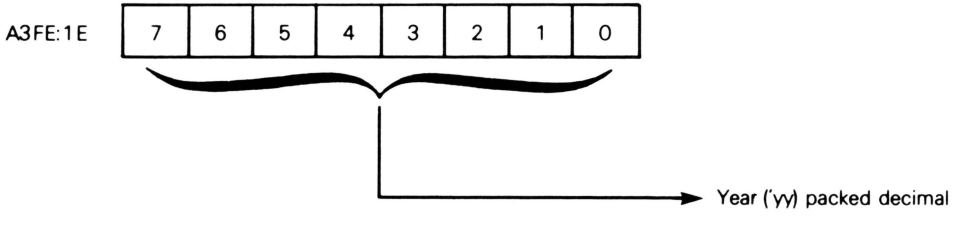
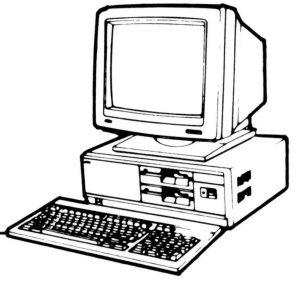


Figure A-8 Year

Appendix B
ANSI.SYS Escape Sequences



ANSI Cursor Control Escape Sequences

Conventions

The following conventions are used throughout this appendix.

1. The default value is used when no explicit value is given or a value of zero is used.

2. Pn is used to specify a numeric parameter; for example, a decimal number specified with ASCII digit(s).

3. Ps is used to specify a selective parameter. For example, any decimal number that is used to select a subfunction. Multiple subfunctions may be selected by separating the parameters with semicolons (;).

Cursor Functions

CUP — Cursor Position

ESC [P l ; Pc H

HVP — Horizontal & Vertical Position

ESC [P l ; Pc f

CUP and HVP move the cursor to the position specified by the parameters. The first parameter specifies the line number and the second parameter specifies the column number. The default value is one. When no parameters are supplied the cursor moves to the home position.

CUU — Cursor Up

ESC [Pn A

CUU moves the cursor up one or more lines without changing columns. The default value of Pn is one. This sequence is ignored if the cursor is already positioned on the top line.

CUD — Cursor Down

ESC [Pn B

CUD moves the cursor down one or more lines without changing columns. The default value for Pn is one. This sequence is ignored if the cursor is already on the bottom line.

CUF — Cursor Forward

ESC [Pn C

CUF moves the cursor forward one or more columns without changing lines. The default value for Pn is one. This sequence is ignored if the cursor is already in the rightmost column.

CUB — Cursor Backward

ESC [Pn D

CUB moves the cursor backward one or more columns without changing lines. The default value for Pn is one. This sequence is ignored if the cursor is already in the leftmost column.

DSR — Device Status Report

ESC [6 n

The console driver will output a CPR sequence on receipt of DSR.

CPR — Cursor Position Report

ESC [Pn ; Pn R

The CPR sequence reports current cursor position via standard input. The first parameter specifies the current line and the second parameter specifies the current column.

SCP — Save Cursor Position

ESC [s

The current cursor position is saved. This position can be restored with the RCP sequence.

RCP — Restore Cursor Position

ESC [u

RCP restores the cursor position to the value it had when the console driver received the SCP sequence.

Erasing Functions

ED — Erase Display

ESC [2 J

Erases all of the screen and the cursor returns to home position.

ESC [J or ESC [0 J

Erases from the cursor to the end of the screen, including the cursor position.

ESC [1 J

Erases from the start of the screen to the cursor, including the cursor position.

EL — Erase Line

ESC [K or ESC [0 K

Erases from the cursor to the end of the line, including the cursor position.

ESC [1 K

Erases from the start of the line to the cursor position, including the cursor position.

ESC [2 K

Erases all of the line including the cursor position.

Modes of Operation

SGR — Set Graphics Rendition

ESC [Ps ; ... ; Ps m

SGR invokes the graphic rendition specified by the parameter(s). The following characters in Table B-1 are rendered according to the parameter(s) until the next occurrence of SGR.

Table B-1 SGR Parameters

Parameter	Parameter Function	
0	All attributes off	
4	Underscore on	
5	Blink on	
7	Reverse video on	
8	Concealed on	(ISO 6429 standard)
30	Black foreground	(ISO 6429 standard)
31	Red foreground	(ISO 6429 standard)
32	Green foreground	(ISO 6429 standard)
33	Yellow foreground	(ISO 6429 standard)
34	Blue foreground	(ISO 6429 standard)
35	Magenta foreground	(ISO 6429 standard)
36	Cyan foreground	(ISO 6429 standard)
37	White foreground	(ISO 6429 standard)

SM — Set Mode

Note The SM escape sequence is not an ANSI standard. It is only applicable to NEC and MicroSoft.

The possible control sequences are:

ESC [= Ps h
ESC [= h
ESC [= 0 h
ESC [? 7 h

The ESC [? sequence applies to end of line only. SM invokes the screen width or type specified by the parameter. Table B-2 lists the available set modes.

Table B-2 Set Modes

Parameter	Parameter Function	Mode
0	40 × 25 black and white	Text
1	40 × 25 color	Text
2	80 × 25 black and white	Text
3	80 × 25 color	Text
4	320 × 200 color	Graphics
5	320 × 200 black and white	Graphics
6	640 × 200 black and white	Graphics
7	Wrap end of line	
8	640 × 200 color	Graphics
9	640 × 400 black and white	Graphics
10	640 x 400 color	Graphics

ESC [> 5 h

Enables cursor display.

RM — Reset Mode

The possible control sequences are:

ESC [= Ps l
ESC [= l
ESC [= 0 l
ESC [? 7 l

Parameters are the same as SM (Set Mode) except that parameter 7 will reset wrap at end of line mode. The character "l" is a lower case letter and not the number 1.

ESC [> 5 l

Disables cursor display.

Keyboard Reassignment

Although the APC III keyboard reassignment is not part of the standard ANSI 3.64-1979 and ISO 6429, it is performed in a compatible manner.

The possible control sequences are:

ESC [Pn ; Pn ; ... Pn p
ESC ["string" ; p
ESC [Pn ; "string" ; Pn ; Pn ; "string" ; Pn p

or any other combinations of string and decimal numbers.

The final code in the control sequence ("p") is reserved for private use by the ANSI 3.64-1979 standard.

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.

There is one exception. If the first code in the sequence is zero (NUL), then the first and second code make up an extended ASCII redefinition. Refer to Appendix H for a list of all the ASCII and extended ASCII codes.

Following are two examples of keyboard reassignment using control sequences.

1. Reassign the Q and q key to the A and a key, and then reassign them to their original values.

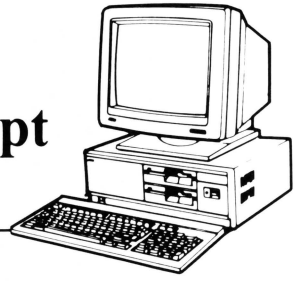
ESC [6 5 ; 8 1 p	A becomes Q
ESC [9 7 ; 1 1 3 p	a becomes q
ESC [8 1 ; 6 5 p	Q becomes A
ESC [1 1 3 ; 9 7 p	q becomes a

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

ESC [0 ; 1 0 7 ; " d i r " ; 1 3 p

The 0 ; 1 0 7 is the extended ASCII code for the F10 key; 13 decimal is a carriage return.

Appendix C
APC III/MS-DOS Interrupt Vector Assignments



The 8086 processor reserves the lowest 1024 bytes of memory for 256 interrupt vectors. Each interrupt vector contains a double word (4-byte) address consisting of an offset followed by a segment. The interrupt vector contents are used by the processor as the address of the routine assigned to handle an interrupt.

Interrupt Levels

Many interrupt levels (and the vectors associated with them) are reserved for special uses in the APC III. The 8086 processor, APC III hardware, ROM BIOS, MS-DOS operating system, GSX, and GW BASIC Interpreter reserve some of the interrupt levels.

The 8259 interrupt controller level 0 interrupt is attached to the 8086 interrupt level 8, 8259 level 1 is attached to the 8086 level 9, and so forth. Note that the 8259 interrupt levels are mapped into the 8086 interrupt levels starting with 8086 level 8 (timer interrupt). Table C-1 lists the interrupt levels and their reserved uses.

Table C-1 Interrupt Levels

Level (hex)	Vector Address (hex)	Reserved Function
00H	0000:0000	Divide by zero (8086)
01H	0000:0004	Single step (8086)
02H	0000:0008	Non-maskable interrupt (NMI) (8086)
03H	0000:000C	Breakpoint instruction (8086)
04H	0000:0010	Arithmetic overflow (8086)
05H	0000:0014	CRTDUMP activation (MS-DOS CRTDUMP)
06H	0000:0018	Gets control on keyboard break*
07H	0000:001C	Gets control on timer interrupt*
08H	0000:0020	Timer interrupt
09H	0000:0024	Keyboard interrupt
0AH	0000:0028	CRT interrupt**
0BH	0000:002C	Extended bus INT0**
0CH	0000:0030	RS-232 interrupt
0DH	0000:0034	Extended bus INT1**
0EH	0000:0038	Extended bus INT2**
0FH	0000:003C	Reserved**
010	0000:0040	Printer interrupt**
011	0000:0044	Extended bus INT3**
012	0000:0048	Diskette interrupt
013	0000:004C	Extended bus INT4**
014	0000:0050	Extended bus INT5**
015	0000:0054	Extended bus INT6**
016	0000:0058	8087 APU interrupt
017	0000:005C	Reserved**
018	0000:0060	Keyboard I/O call (ROM BIOS)
019	0000:0064	CRT I/O call (ROM BIOS)
01A	0000:0068	RS-232C I/O call (ROM BIOS)
01B	0000:006C	Printer I/O call (ROM BIOS)
01C	0000:0070	Timer I/O call (ROM BIOS)
01D	0000:0074	Disk I/O call (ROM BIOS)
01E	0000:0078	OPTRS232 I/O call
01F	0000:007C	Reserved
020	0000:0080	MS-DOS program terminate
021	0000:0084	MS-DOS function call
022	0000:0088	MS-DOS terminate address
023	0000:008C	MS-DOS CTRL-BREAK exit address

Table C-1 Interrupt Levels (cont'd)

Level (hex)	Vector Address (hex)	Reserved Function
024	0000:0090	MS-DOS fatal error vector
025	0000:0094	MS-DOS absolute disk read
026	0000:0098	MS-DOS absolute disk write
027	0000:009C	MS-DOS terminate, fix in storage
028	0000:00A0	Background process entry
029	0000:00A4	Reserved for MS-DOS
.	.	.
.	.	.
.	.	.
03F	0000:00FF	Reserved for MS-DOS
040	0000:0100	Not used
.	.	.
.	.	.
.	.	.
0E0	0000:0380	GSX function call
.	.	.
.	.	.
.	.	.
0FF	0000:03FF	Not used

*Initialized at power up to point to an IRET instruction.

**Initialized at power up to point to an EOI process.

Extended Bus Interrupt Vectors

Many of the APC III expansion options use the extended bus interrupt vectors. Table C-2 shows the INTs that are available for each option. In order to avoid conflicts, you should make sure that each option operates on a unique INT; that is, a particular INT is not used for more than one option. Most of the options can choose from among several INTs, but for the hard disk controller, only one choice is available. Factory settings are indicated, where applicable.

Table C-2 Extended BUS INT Usage

Option and Kit Number	INT0	INT1	INT2	INT3	INT4	INT5	INT6
Joystick/Sound Card APC-H152	—	—	X	—	X	X	X*
General Purpose Interface Bus APC-H150	X	—	—	—	X	X	X
Additional Serial Interface Card APC-H156							
First Optional Port (AUX2)	X	X*	X	X	—	—	—
Second Optional Port (AUX3)	X*	—	—	—	X	X	X
10 MB Hard Disk APC-H112, APC-H170, APC-H171	—	—	—	X**	—	—	—
PC-UX Memory Management Card APC-H155	X	—	—	—	X	X	X*

* Indicates factory setting.

** Factory set, cannot be changed.

Vectors with Special Meanings

Following are explanations of vectors that have special meanings.

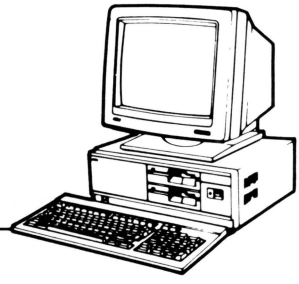
INTERRUPT 06H — Keyboard Break Address

The Keyboard Break Address points to the code to be exercised when you press <Ctrl> and <Break Stop>. The vector is invoked while responding to the keyboard interrupt. Control should be returned via the IRET instruction. The power on routines initialize this vector to point to an IRET instruction. This ensures that nothing happens when you press <Ctrl> and <Break Stop> unless the application program sets a different value. Control may be retained by this routine, if the break occurred during interrupt processing.

INTERRUPT 07H — Timer Internal Process Address

The timer internal process address vector points to the code to be executed after every time interval. This vector is invoked while responding to the TIMER I/O. Control should be returned via an IRET instruction. The power on routines initialize this vector to point to an IRET instruction. This ensures that nothing happens unless the application modifies the pointer. It is the application's responsibility to save and restore all registers to be modified.

Appendix D
CONFIG.SYS Configuration File



Configuration Files in MS-DOS V. 2.0 and Higher

In many cases, there are installation-specific configurations of the DOS that need to be set up at boot time. It is considered a poor design for a user/OEM to have to rebuild MS-DOS in order to include special device drivers, or a number of device drivers. The configuration file allows you to configure the system without extra work.

The configuration file is simply an ASCII file that has certain commands for the bootstrap. The bootstrap process follows.

1. The disk boot sector is read. The sector contains enough code to read the DOS and initial BIOS.

2. The DOS and initial BIOS are read.

3. A long jump is made to the BIOSINIT routine. A variety of BIOS initializations are performed.

4. A long jump is made to the SYSINIT routine in the SYSINIT module. This module (supplied by Microsoft) initializes the DOS and reads the CONFIG.SYS file. If CONFIG.SYS exists, it performs device installations and other user-settable options.

5. Its final task is to EXEC the command interpreter, which completes the bootstrap process.

CONFIG.SYS Commands

The following is a list of commands that can appear in the file CONFIG.SYS.

BUFFERS = <number>

This is the number of additional sector buffers to add to the system list. The effect of several BUFFERS commands is to allocate a series of buffers.

FILES = <number>

This is the number of open files that the XENIX system calls can access.

DEVICE = <filename>

This installs the device driver in <filename> into the system list.

BREAK = <ON or OFF>

If ON is specified (the default is OFF), a check for ^C at the console input will be made every time the system is called. ON improves the ability to abort programs over previous versions of the DOS.

SHELL = <filename>

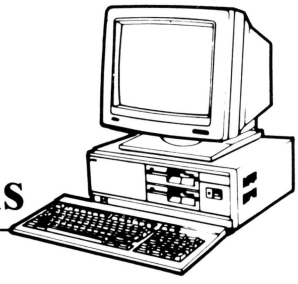
This begins execution of the shell (top-level command processor) from <filename>.

A typical configuration file might look like this:

```
BUFFERS = 10
FILES = 10
DEVICE = \bin\network.sys
BREAK = ON
SHELL = a:\bin\command.com a:\bin /p
```

The default value for BUFFERS in the APC III I/O system is 2. The minimal value is one. The default value for FILES is 8, so "FILES = 10" would actually allocate only two new file channels. If a number less than or equal to 5 is specified, the command is ignored. The default value for BREAK is off.

Appendix E
ROM BIOS Function Calls



This Appendix gives a summary of all the ROM BIOS calls available to the programmer. For more extensive information on ROM BIOS calls, refer to the *APC III System Reference Guide* and the ROM BIOS lists available on diskette.

Note Programmers should not presume any register preservation or make ROM BIOS calls with undocumented parameters, as the results are unpredictable.

KEYBOARD_IO Call

KEYBOARD_IO 18H

Calling Sequence MOV AH, FUNCTION; (see below for subfunction numbers).

INT KEYBOARD_IO ; 18H

Input

(AH)=0 Read the next ASCII character from the keyboard FIFO queue.

Return with the result in (AL), scan code in (AH). If AL is 0, then the Extended ASCII code is returned in AH.

(AH)=1 Set the Z flag to indicate whether an ASCII character is available to be read from the FIFO queue.

(ZF)=1 No code is available. The queue is empty.

(ZF)=0 Code is available. There is at least one character in the queue. The next character read from the FIFO queue is returned in AL and the key's scan code is returned in AH unless the key produces an extended ASCII code. In this case, AH will contain the second code returned for the key and AL will always contain 00H.

(AH)=2 Returns the current shift status in the AL register. The bit settings for this code are as follows:

bit 7(MSB)=1	Insert state is active.
bit 6 =1	<Caps Lock> state has been toggled.
bit 5=1	<Grph 1> lock state has been toggled.
bit 4=1	<Grph 2> lock state has been toggled.
bit 3=1	<Alt> shift key pressed.
bit 2=1	<Ctrl> shift key pressed.
bit 1=1	<Shift> key pressed.
bit 0(LSB)=1	<Fnc> shift key pressed.

CRT_IO Call

CRT_IO 19H

Calling Sequence
```
MOV AH, FUNCTION
INT CRT_IO ; 19H
```

Note In the CRT_IO functions, registers CS, SS, DS, ES, BX, CX, and DX are preserved in the call. All other registers are destroyed.

Input

(AH)=0 Set a display mode. (AL) contains the mode value. The following modes are available.

Text modes

(AL)=0 40 × 25 monochrome (master GDC)
(AL)=1 40 × 25 color (master GDC)
(AL)=2 80 × 25 monochrome (master GDC)
(AL)=3 80 × 25 color (master GDC)

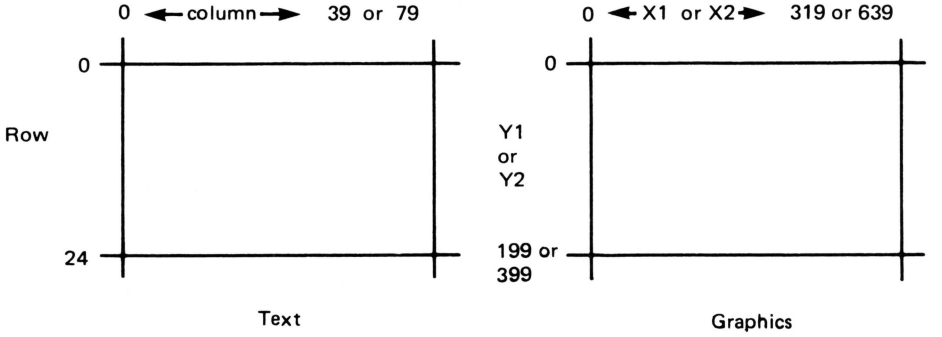
Graphic modes

(AL)=4 320 × 200 color (slave GDC)
(AL)=5 320 × 200 monochrome (slave GDC)
(AL)=6 640 × 200 monochrome (slave GDC)
(AL)=7 640 × 200 color (slave GDC)
(AL)=8 640 × 400 monochrome (slave GDC)
(AL)=9 640 × 400 color (slave GDC)

Whenever a new mode is selected, the entire screen is cleared. Modes 8 and 9 are used for CRT monitor displays only. CRT_IO does not care what the present CRT mode is. If a color graphics mode is specified, CRT_IO checks the color graphics board (VRAM). If there is no color graphics board, the mode is changed automatically from color to black and white. If modes 5, 6, or 8 are specified, CRT_IO sets the palette color ID to the fixed value.

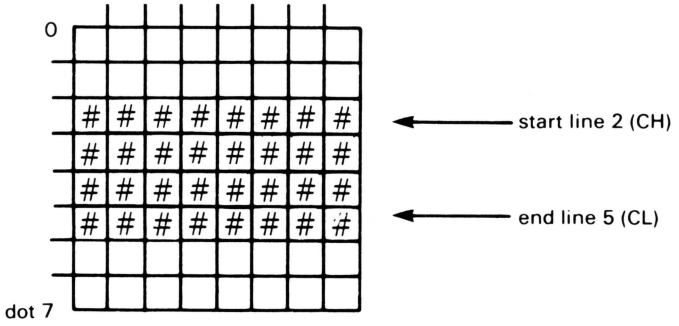
Note After making this call, users must delay writing to text or graphics VRAM for approximately 65 milliseconds to avoid corruption of displayed data.

GDC has two screen pages in both text and graphics modes. The row and column positions are defined as follows.



(AH)=1 Set cursor type.

(CH)=2 and (CL)=5, the following cursor type will be displayed.



(CH) and (CL) must be below 7.

If bit 4 of the CH register is set to 1, the cursor will not blink. If the MSB (bit 7) of CH is set to 1, the cursor display will be disabled.

(AH)=2 Set cursor position

(DH, DL) = Row, column; (0,0) is upper left

(BH) = Page number, (0,1)

(AH)=3 Read cursor position

(BH) = Page number, (0,1)

On exit,

(DH), DL) = Row, column of current cursor.

(CH), CL) = Cursor type (same as set cursor type function).

(AH)=4 Reserved for future use.

(AH)=5 Select active display page.

(AL) = New page value (0 or 1).

(AH)=6 Scroll active page up.

(AL) = Number of lines, input lines blanked at bottom of window. AL=0 indicates blank entire window.

(BH) = Attribute to be used on blank line (text mode). Color ID to be used on blank line (graphics mode).

(CH, CL) = Row, column of upper left corner of window to scroll.

(DH), DL) = Row, column of lower right corner of window to scroll.

A space code is used on blank lines in the text modes.

(AH)=7 Scroll active page down

(AL) = Number of lines, input lines blanked at top of window. AL=0 indicates blanked at top of window.

(BH) = Attribute to be used on blank line (text mode). Color ID to be used on blank line (graphics mode).

(CH, CL) = Row, column of upper left corner of window to scroll.

(DH, DL) = Row column of lower right corner of window to scroll.

Character Handling Functions

(AH)=8 Read attribute/character at current cursor position.

(BH) = Display page.

On exit,

(AL) = Character read.

(AH) = Attribute of character read (text).

The current cursor position is not changed.

(AH)=9 Write attribute/character at current cursor position.

(BH) = Display page.

(CX) = Count of characters to write.

(AL) = Character to write.

(BL) = Attribute of character (text).
= Palette number (graphics).

The palette number includes the operational codes as shown in Figure E-1.

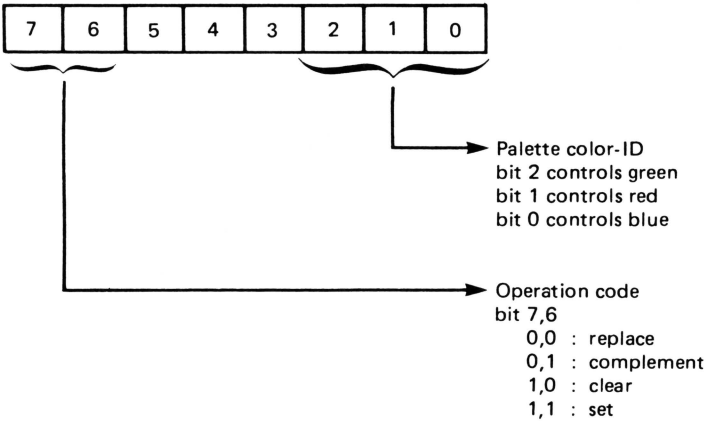


Figure E-1 Palette Numbers and Codes

(AH)=10 Write character only at current cursor position.

(BH) = Display page.

(CX) = Count of characters to write (not equal to 0).

(AL) = Character to write.

(BL) = Palette number (graphics).

The replication factor contained in (CX) will only produce valid results for characters on the same row in the graphics writer character interface. Continuation to succeeding lines will be produced incorrectly.

Graphics Interface

(AH)=11 Set color palette

(BH) = Palette color ID being set (0-7).

(BL) = Color value used with color ID (0-7).

(AH)=12 Write dot.

(DX) = Row number.

(CX) = Column number.

(AL) = Palette number.

Bit 7 and 6 of AL are used as MOD for the μPD7220 GDC:

00 - replace
01 - complement
10 - clear
11 - set

(AH)=13 Read dot.

(DX) = Row number.

(CX) = Column number.

On exit,

(AL) = Returns palette number. In monochrome modes (AL) = 0 or 1, black is 0.

ASCII teletype routine for output

(AH)=14 Write teletype.

(AL) = Character to write.

(BL) = Palette number (graphics only).

The available ASCII control characters are as follows:

07h → Bell
08h → Back Space
0Ah → Line Feed
0Dh → Carriage Return

In the text modes, the space code is used as a filling character when you scroll. Previous cursor position's attribute is used as the attribute of the filling character.

In graphic modes, the current active display page is used (BH). The lowest three bits used for palette numbers are 0,0,0.

(AH)=15 Current CRT state.

On exit,

(AL) = Mode currently set.

(AH) = Number of character columns on screen.

(BH) = Current active display page.

CS, SS, DS, ES, BX, CX, and DX are preserved during the call. All others are destroyed.

Slave GDC Functions

Note The following functions are not supported in the 320 × 200 character graphics mode.

(AH)=16 Set draw mode function

(AL) = 0 : Flash draw.

(AL) = 1 : Flashless draw (default mode).

(AH)=17 Set draw page function

(AL) = Page number (0 or 1; default is 0). The specified page is used in the following functions.

(AL) = 12 : Write dot functions.
(AL) = 13 : Read dot functions.
(AL) = 18 : Clear graphic screen function.
(AL) = 19 : Draw line function.
(AL) = 20 : Draw rectangle function.
(AL) = 21 : Tile rectangle function.
(AL) = 22 : Draw circle function.

(AH)=18 Clear graphics screen function

(AL) = Palette number (not included in operational code).

(AH)=19 Draw line function

(AL) = Palette number.

(BX) = Line pattern (must be a 16-bit pattern).

(SI, DI) = (X1,Y1) point 1

(CX, DX) = (X2,Y2) point 2

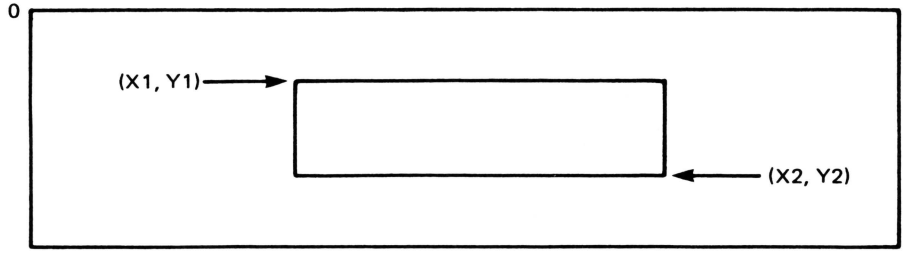
(AH)=20 Draw rectangle function

(AL) = Palette number.

(BX) = Line pattern.

(SI, DI) = (X1,Y1) point 1

(CX, DX) = (X2,Y2) point 2

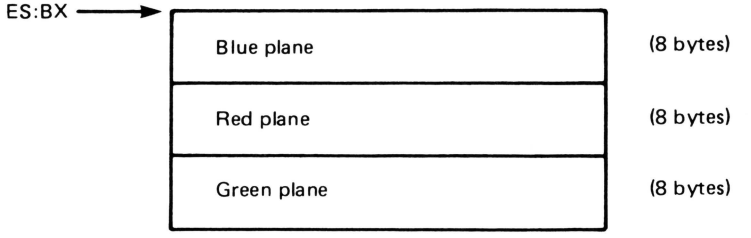


(AH)=21 Tile rectangle function

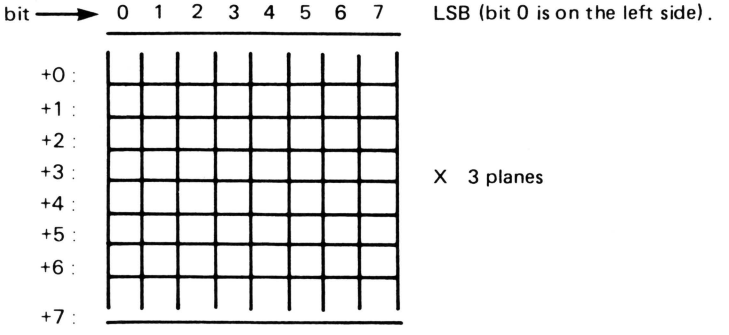
(SI, DI) = (X1,Y1) point 1

(CX, DX) = (X2,Y2) point 2

(ES:BX) = Tile pattern pointer



Tile pattern (each plane has 8 bytes)



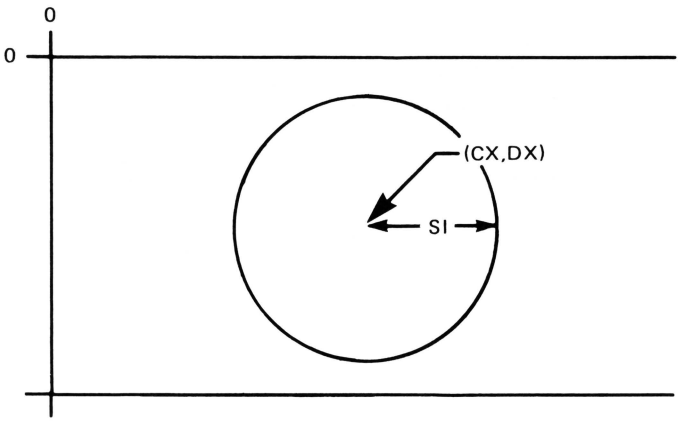
(AH)=22 Draw circle function

(AL) = Palette number.

(BX) = Line pattern.

(SI) = Radius.

(CX, DX) = Center of circle (X1,Y1).



In the 640 × 200 character modes, an ellipse is drawn. Also, complement operation code in (AL) is not supported.

RS232C_IO Call

RS232C_IO 1AH

Calling Sequence
```
MOV AH, FUNCTION
INT RS232C_IO ; 1 AH
```

Input

(AH)=0 Initialize RS-232C.

(AL) Baud Rate,

0 - 75 bps
1 - 150 bps
2 - 300 bps
3 - 600 bps
4 - 1200 bps
5 - 2400 bps
6 - 4800 bps
7 - 9600 bps

(CH) Asynchronous mode for μPD 8251A

bit 7, 6 : Stop bit length
01 - 1 bit
10 - 1.5 bit
11 - 2 bit

bit 5, 4 : Parity
XO - none
01 - odd
11 - even
bit 3, 2 : Word length
00 - 5 bits
01 - 6 bits
10 - 7 bits
11 - 8 bits
bit 1, 0 : Clock
10 - x16
11 - x64

(CL) Command for μPD8251A

bit 6 : Interrupt reset
bit 5 : Request to send
bit 4 : Error request
bit 3 : Send break character
bit 2 : Receive enable
bit 1 : DTR on
bit 0 : Send enable

(ES:DI) Receive buffer pointer

(DX) Receive buffer byte length

(BH) Time out second on send; a value of 0 sets the default.

(BL) Time out second on receive; a value of 0 sets the default.

Note The selection of 0 (default) time out values in the initialize RS-232C function calls (set in registers BH and BL) will result in the selection of "infinite" (no) time out values for the RS232C_IO ROM BIOS functions.

On exit,

(AH) Return Code
0 - Normal exit
1 - Not available
2 - Receive buffer overflow
3 - Send disable/receive disable

(AH)=1 Initialize RS-232C for X ON/OFF protocol.

AL, CH, CL, ES:DI, DX, BH, BL, (same as initialize RS-232C (AH)=0).

On exit,

(AH) Return code same as initialize RS-232C.

(AH)=2 Get data length in receive buffer.

On exit,

(AH) Return code.

(CX) Data length.

(AH)=3 Send a character in (AL).

On exit,

(AH) Return code same as initialize RS-232C.

(AH)=4 Receive character from receive buffer.

On exit,

(AH) Return code

(CH) Receive character

(CL) Line status on receive
bit 7 : Data set ready
bit 6 : SYNC detect
bit 5 : Framing error
bit 4 : Data overrun
bit 3 : Parity error
bit 2 : Transmitter empty
bit 1 : Clear to send
bit 0 : Carrier detect

(AH)=5 Command for μPD8251A.

(AL) μPD8251A (see the Initialize RS232-C (CL)

On exit,

(AH) Return code

(AH)=6 Get status in (CH) and (CL).

On exit,

(AH) Return code

(CH) μPD8251A status
bit 7 : Data set ready
bit 6 : SYNC detect
bit 5 : Framing error
bit 4 : Data overrun
bit 3 : Parity error
bit 2 : Transmitter empty
bit 1 : Receive ready
bit 0 : Transmit ready

(CL) Port status.
bit 7 : Ring indicator 0: activated
1: deactivated
bit 6 : Clear to send 0: enabled
1: disabled
bit 5 : Carrier detect 0: detected
1: nondetected

PRINTER_IO Call

PRINTER_IO 1BH

Calling Sequence MOV AH, FUNCTION
INT PRINTER_IO ; 1BH

Input

(AH)=0 Print a character in (AL)

On exit,

(AH) Status

(AH)=1 Initialize the printer port.

On exit,

(AH) Status

bit 1, 0
00 - printer ready
01 - printer not ready
10 - time out
11 - power off or not on-line

(AH)=2 Read the printer status into (AH).

(AH)=3 Print n character in data buffer.

(ES:BX) Data buffer pointer.

(CX) Print data length

On exit,

(AH) Status.
(AH)=0 for normal end.

(BX) Data pointer offset at time out state.

(CX) Remaining data length.

(BX) and (CX) are set if return status (AH) is not equal to 0.

TIMER_IO Call

TIMER_IO 1CH

Calling sequence
```
MOV   AH, FUNCTION
INT   TIMER_IO ; 1CH
```

Input

(AH)=0 Get date and time.

(ES:BX) Date and time data buffer pointer.

On exit,

date and time into data buffer

0	Year	Packed decimal value
1	Month	Binary, and stored in upper 4 bits of the byte. November = BXH
2	Day	
3	Hour	
4	Minute	
5	Second	

(AH)=1 Set date and time.

(ES:BX) Date and time data buffer pointer. Same as (AH)=0.

(AH)=2 Set interval timer.

(CX) Interval time (10 msec. ~ 65536 msec.)

(ES:BX) time out interrupt pointer.

DISK_IO Call

DISK_IO 1DH

Calling sequence

```
MOV   AH, COMMAND_OPTION
MOB   AL, DEV_UNIT_OPTION
MOV   BX, DATA_LENGTH
MOV   CX, CYLINDER_SECTOR
MOV   DH, HEAD_NUMBER
MOV   DL, SECTOR_NUMBER
MOV   ES, (buffer segment)*
      BP, (buffer offset)*
INT   DISK_IO ; 1 DH
```

*If you specify an odd address for I O buffer offset, the DMA transfer will be slower. To avoid this specify an even memory address for I O buffer offset. The DMA operation is limited to within a 64 K byte memory bank. Check that the buffer does not exceed the 64 K byte boundary.

Command options

(AH)
bit 7 : Track access option
- 0 - single track
- 1 - multitrack

bit 6 : Density option
- 0 - single density
- 1 - double density

bit 5 : Error retry on track option
- 0 - 8 retry
- 1 - no retry

bit 4 : Seek option
- 0 - no seek performed
- 1 - seek before command

bit 3, 2, 1, 0;
- 0000 - no operation
- 0001 - verify data
- 0010 - read diagnostics
- 0011 - initialize
- 0100 - sense
- 0101 - write data
- 0110 - read data
- 0111 - recalibrate
- 1000 - no operation
- 1001 - write deleted data
- 1010 - read ID
- 1011 - no operation
- 1100 - read deleted area
- 1101 - write ID
- 1110 - no operation
- 1111 - retract (hard disk)

(AL) Device drive unit and option

Option:
bit 7 : Relative sector access on hard disk.
bit 6 : Double-track access on diskette.
Disk type:
bit 5, 4
- 00 - 5¼ Diskette
- 01 - 5¼ Hard disk

bit 3, 2, 1, 0; Unit number.
- 0000 - Unit number 0
- 0001 - Unit number 1

(CH) Sector length for diskette.
0 - 128 byte/sector
1 - 256 byte/sector
2 - 512 byte/sector
3 - 1024 byte/sector

(CL) Cylinder number for diskette.

(DH) Head number for diskette.

(DL) Sector number for diskette.

Hard disk parameters
If bit 7 of (AL) = 1
  (CX) Relative sector number (low)
  (DX) Relative sector number (high) for hard disk
If bit 7 of (AL) = 0
  (CX) Track number
  (DX) Sector number for hard disk

On exit,

(C-flag) First status.
- 0 - Successful end
- 1 - Unsuccessful end

Second status (AH)

bit 7, 6, 5, 4; Access status.

0000 - Normal end (C-flag = 0)
0001 - Control mark (C-flag = 0)
0010 - DMA boundary (C-flag = 1)
0011 - End of cylinder (C-flag = 1)
0100 - Not ready (C-flag = 1)
0101 - Not writable (C-flag = 1)
0110 - Equipment check (C-flag = 1)
0111 - Equipment check on seek (C-flag = 1)
1000 - Over run (C-flag = 1)
1001 - Time out (C-flag = 1)
1010 - Data error (ID) (C-flag = 1)
1011 - Data error (data) (C-flag = 1)
1100 - No data (C-flag = 1)
1101 - Bad cylinder (C-flag = 1)
1110 - Missing address (ID) (C-flag = 1)

bit 1, 0; Hard disk type on sense command to hard disk
01-10 MB hard disk

bit 0 ; Diskette type on sense command to FD.
0 - Double-sided double-density (2D)

OPTRS232C_IO Call

OPTRS232C_IO 1EH

Calling Sequence
```
MOV AH, FUNCTION
INT OPTRS232C_IO ; 1EH
```

Input For all calls, (SI) indicates the port being used.

(SI)
0 - Use AUX port (pass call to INT 1A)
1 - Use AUX2 port
2 - Use AUX3 port

(AH)=0 Initialize RS-232C.

(AL) Baud Rate,

0 - 75 bps
1 - 150 bps
2 - 300 bps
3 - 600 bps
4 - 1200 bps
5 - 2400 bps
6 - 4800 bps
7 - 9600 bps

(CH) Asynchronous mode for μPD8251A

bit 7, 6 : Stop bit length
- 01 - 1 bit
- 10 - 1.5 bit
- 11 - 2 bit

bit 5, 4 : Parity
- XO - none
- 01 - odd
- 11 - even

bit 3, 2 : Word length
- 00 - 5 bits
- 01 - 6 bits
- 10 - 7 bits
- 11 - 8 bits

bit 1, 0 : Clock
- 10 - x16
- 11 - x64

(CL) Command for μPD8251A

bit 6 : Interrupt reset
bit 5 : Request to send
bit 4 : Error request
bit 3 : Send break character
bit 2 : Receive enable
bit 1 : DTR on
bit 0 : Send enable

(ES:DI) Receive buffer pointer

(DX) Receive buffer byte length

(BH) Time out second on send; a value of 0 sets the default.

(BL) Time out second on receive; a value of 0 sets the default.

Note The selection of 0 (default) time out values in the initialize RS-232C function calls (set in registers BH and BL) will result in the selection of "infinite" (no) time out values for the OPTRS232C_IO ROM BIOS functions.

On exit,

(AH) Return Code
0 - Normal exit
1 - Not available
2 - Receive buffer overflow
3 - Send disable/receive disable

(AH)=1 Initialize RS-232C for X ON/OFF protocol.

AL, CH, CL, ES:DI, DX, BH, BL, (same as initialize RS-232C (AH)=0).

On exit,

(AH) Return code same as initialize RS-232C.

(AH)=2 Get data length in receive buffer.

On exit,

(AH) Return code.

(CX) Data length.

(AH)=3 Send a character in (AL).

On exit,

(AH) Return code same as initialize RS-232C.

(AH)=4 Receive character from receive buffer.

On exit,

(AH) Return code

(CH) Receive character

(CL) Line status on receive
bit 7 : Data set ready
bit 6 : SYNC detect
bit 5 : Framing error
bit 4 : Data overrun
bit 3 : Parity error
bit 2 : Transmitter empty
bit 1 : Clear to send
bit 0 : Carrier detect

(AH)=5 Command for μPD8251A.

(AL) μPD8251A (see the Initialize RS232-C (CL))

On exit,

(AH) Return code

(AH)=6 Get status in (CH) and (CL).

On exit,

(AH) Return code

(CH) μPD8251A status
bit 7 : Data set ready
bit 6 : SYNC detect
bit 5 : Framing error
bit 4 : Data overrun
bit 3 : Parity error
bit 2 : Transmitter empty
bit 1 : Receive ready
bit 0 : Transmit ready

(CL) Port status.
bit 7: Ring indicator 0: activated
1: deactivated
bit 6: Clear to send 0: enabled
1: disabled
bit 5: Carrier detect 0: detected
1: nondetected
bit 1: IL1 (see chart for values)
bit 0: IL2 (see chart for values)

		Interrupt Assignment	
IL1	**IL2**	**AUX2**	**AUX3**
0	0	INT0 (IR3)	INT0 (IR3)
0	1	INT1 (IR5)	INT4 (IR11)
1	0	INT2 (IR6)	INT5 (IR12)
1	1	INT3 (IR9)	INT6 (IR13)

Appendix F
CRT Display Character Set

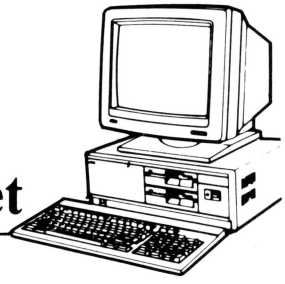


Figure F-1 illustrates the displayable character set for the APC III. The two digit hexadecimal code for any character can be found by taking the number in the top row as its first digit, and the number in the left column as its second digit. The characters are defined by the contents of an internal hardware character-generator ROM. The figure defines the graphics that display when the corresponding code is written into a character position in the VRAM memory.

Certain characters cannot be displayed through the CON device driver because they have special meanings. For example, the character code 13 (decimal) is reserved for Carriage Return. The console driver will intercept the character, and act on it, rather than place it into VRAM.

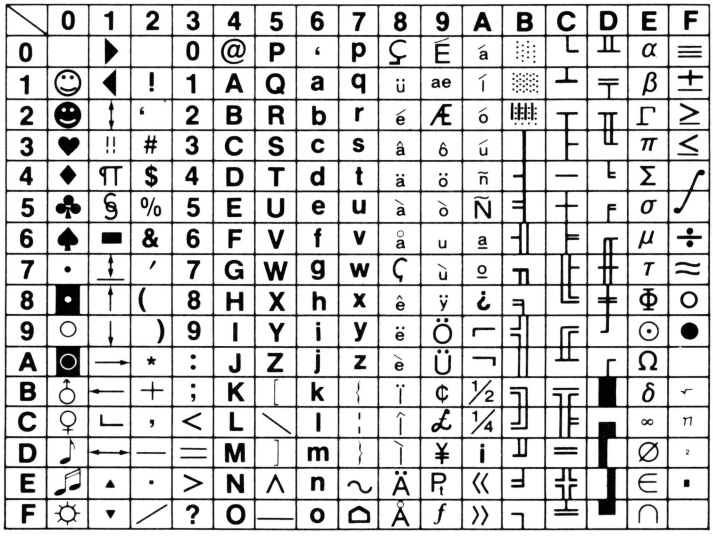
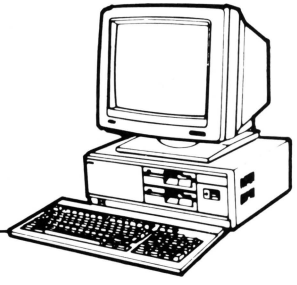


Figure F-1 The APC III Displayable Character Set

Appendix G
VRAM CRT Display



Text VRAM Memory Map

The standard APC III is equipped with 8 K bytes of standard text VRAM. Table G-1 lists how the memory address space is situated in the 8086 MPU.

Table G-1 MPU Memory Address Space

MPU Address	Area
A000:0000-A000:0F9F	Page 1 character codes
A000:1000-A000:1F9F	Page 2 character codes
A000:2000-A000:2F9F	Page 1 character attributes
A000:3000-A000:3F9F	Page 2 character attributes

Within each display area, the mapping of MPU memory address to screen positions is listed in Table G-2. The examples use the address space for Page 1 character codes, but the mapping also applies to the other regions (Page 2 characters and attributes).

Table G-2 Mapping of MPU Memory Addresses

MPU Address	Text Display Mode 80 × 25	40 × 25	
A000;0000	(1, 1)	(1, 1)	(displayed char. pos.)
A000;0001	x	x	x = unused
A000;0002	(1, 2)	x	
A000;0003	x	x	
A000:0004	(1, 3)	(1, 2)	
A000:0005	x	x	
A000:0006	(1, 4)	x	
A000:0007	x	x	
A000:0008	(1, 5)	(1, 3)	
A000:0009	.	.	
	.	.	
	.	.	
A000:009C	(1, 79)	(1, 40)	
A000:009D	x	x	
A000:009E	(1, 80)	x	
A000:009F	x	x	
A000:00A0	(2, 1)	(2, 1)	
A000:00A1	x	x	
A000:00A2	(2, 2)		
A000:00A3	x	x	
A000:00A4	(2, 3)	(2, 2)	
A000:00A5	x	x	
A000:00A6	.	.	
	.	.	
	.	.	
A000:0F9C	(25, 79)	(25, 40)	
A000:0F9D	x	x	
A000:0F9E	(25, 80)	x	
A000:0F9F	x	x	

Text Mode Attributes

Each character in the APC III text VRAM has a corresponding attribute byte (located 2000H bytes above the character). Each bit in the attribute byte specifies whether the character will be displayed with a particular attribute. By combining bits in the attribute byte, combinations of attributes for a given display character can be achieved.

The format of the attribute byte follows.

Bit 7 (high order)	Green
6	Red
5	Blue
4	Vertical Strike Through Standard Graphics
3	Underline
2	Reverse Video
1	Blinking
0 (low order)	Overline

Color Bits

The color bits (green, red, blue) have different effects on color and monochrome displays. For color displays, they each control the output of one of the display's three color guns. For monochrome displays, the color bits can be used in various combinations to control display intensity. Table G-3 lists the combination of color bits effects for color and monochrome display types.

Table G-3 Color Bit Combinations

Green	Red	Blue	Color	Monochrome
0	0	0	Black	Secret (no display)
0	0	1	Blue	Most dim
0	1	0	Red	.
0	1	1	Magenta	.
1	0	0	Green	.
1	0	1	Cyan	.
1	1	0	Yellow	.
1	1	1	White	Brightest

Standard Graphics Characters

The APC III hardware supports a standard graphics character feature that allows the hardware to map a display character position matrix (8 × 16 pixels) into an array of eight (4 × 4 pixel) tiles. Each of the eight tiles are controlled by a corresponding bit in the display character. To use this feature, a register called "Mode Flip-Flop" must be used to switch the meaning of the Vertical Strike-Through/Standard Graphics attribute bit. The ROM BIOS display modes always set the Mode Flip-Flop to use this bit as vertical Strike-Through. If use of the Standard Graphics Character feature is required, refer to the *APC III System Reference Guide* for more information.

Graphics VRAM Memory Map

The standard APC III is equipped with 64 KB of graphics VRAM. The ROM BIOS includes support for this memory as two pages of a plane of pixels up to 640 × 400 pixel resolution. To achieve color graphics, the APC III Color Graphics option adds memory to a total graphics memory size of 192 KB. This is enough to support two pages of three color planes of up to 640 × 400 pixel resolution. In color modes, the color of a pixel on the display screen is controlled by combining the bits for that pixel position from the three color planes (normally green, red, and blue).

There are two pages (screens) of memory supported by ROM BIOS for every graphics display mode. Additional pages, for lower resolution modes, are allowed by the hardware and may be configured for the software which bypasses the ROM BIOS. A bank switching scheme is used to select the page of the memory to be used for any operation. Also provided is independent control of the page currently accessible by the MPU, in the MPU's memory address space. The ROM BIOS Select Active Page call always selects both of these aspects together.

The correspondence between the three regions of display memory (color planes designated as P1, P2, and P3) and the actual colors that are displayed is controlled by a palette register. The register allows any three-bit input combination to be mapped into a specific combination of the three colors to be displayed. The palette registers can be thought of as an eight-element array (indexed [0..7]) by the three-bit output of the memory color planes. Each element of the array contains a three-bit number that specifies the colors that are sent to the CRT hardware for a corresponding 3-bit input combination.

Note For monochrome displays, the previous discussion is valid, but the colors are displayed as varying shades of grey intensity.

Input	→ Palette Registers →	Output
from color planes	(8 3-bit registers)	to CRT color guns
3 bits: P1, P2, P3		3 bits (R, G, B)

During normal use (including modes set up by ROM BIOS), the palette registers are programmed to make the three color planes behave in the most natural manner; that is, green, red, and blue planes, respectively, as shown in Table G-4.

Table G-4 Palette Registers Input and Output

Input (P1, P2, P3)	Palette Index	Register Contents	Output (G, R, B)	Color
0 0 0	0	0 0 0	0 0 0	black (none)
0 0 1	1	0 0 1	0 0 1	blue
0 1 0	2	0 1 0	0 1 0	red
0 1 1	3	0 1 1	0 1 1	magenta
1 0 0	4	1 0 0	1 0 0	green
1 0 1	5	1 0 1	1 0 1	cyan
1 1 0	6	1 1 0	1 1 0	yellow
1 1 1	7	1 1 1	1 1 1	white

Note For monochrome displays, blue is the dimmest shade, and white is the brightest.

You can alter the palette register using the ROM BIOS calls by modifying the mapping. Table G-5 lists a modified version of Table G-4. The mappings for cyan and magenta have been modified to produce yellow.

Table G-5 Modified Palette Registers Input and Output

Input (P1, P2, P3)	Palette Index	Register Contents	Output (G, R, B)	Color
0 0 0	0	0 0 0	0 0 0	black (none)
0 0 1	1	0 0 1	0 0 1	blue
0 1 0	2	0 1 0	0 1 0	red
0 1 1	3	1 1 0	1 1 0	yellow
1 0 0	4	1 0 0	1 0 0	green
1 0 1	5	1 1 0	1 1 0	yellow
1 1 0	6	1 1 0	1 1 0	yellow
1 1 1	7	1 1 1	1 1 1	white

The remapping can be achieved by simply altering the contents of palette registers 3 and 5.

The APC III can be equipped with 1 (standard) and 3 (color graphics option) planes of graphics display memory. It is sufficient for images of up to 640 × 400 and eight different colors (including black). Each plane consists of 64 KB of memory. Each bit in the memory corresponds to a single pixel on the screen. Table G-6 summarizes the use of graphics on the APC III for the modes available through ROM BIOS. This is not a comprehensive list of the possible modes. For a more detailed list of the graphics display hardware, refer to the *APC III System Reference Guide.*

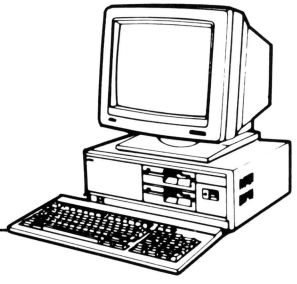
Table G-6 Available Graphic Modes on the APC III

MPU Address Start	End	Mode	ROM BIOS Modes
A800:0000	A800:7D00	P1 640 × 400	8, 9
A800:0000	A800:3E80	P1 640 × 200	6, 7
A800:0000	A800:3E80	P1 320 × 200*	4, 5
B000:0000	B000:7D00	P2 640 × 400	9
B000:0000	B000:3E80	P2 640 × 200	7
B000:0000	B000:3E80	P2 320 × 200*	4
B800:0000	B800:7D00	P3 640 × 400	9
B800:0000	B800:3E80	P3 640 × 200	7
B800:0000	B800:3E80	P3 320 × 200*	4

*The 320 × 200 graphics modes are accomplished in the ROM BIOS by actually selecting a 640 × 200 pixel mode and doubling the pixels in the horizontal direction as they are set. This will be transparent to programs that use the ROM BIOS calls that manipulate pixels. However, any program directly accessing the graphics VRAM in 320 × 200 modes will have to perform the doubling itself. For example, to set logical pixel (0,0) in 320 × 200 mode, the program would need to set BOTH bit 7 and bit 6 at A800:0000.

Within each plane and for every mode, the correspondence between bits of the VRAM memory and pixels are the same. The first raster line of pixels on the screen may come from the first 80 (decimal) bytes of the memory area. The second raster line will come from the next 80 bytes and so forth until the last (200th or 400th) line has been mapped. From each byte of memory, the high-order bit is displayed first.

Appendix H
Keyboard Character Set



This appendix gives a list of the APC III keyboard character set. Figures H-1 and H-2 shows the APC III keyboard layout and the keyboard scan codes/key position conversion chart, respectively. In Figure H-1, the numbers appearing in the lower corner designate the key position.

In Table H-1 the numbers in parentheses are either ASCII or extended ASCII codes. An extended ASCII code would look similar to (00 XX). The codes that are not in parentheses are either graphic or mnemonic symbols associated with the ASCII code. Please note that certain key/shift state combinations which produce valid extended ASCII codes are not listed in Table H-1. These codes are produced for various keys in the available shift states and are listed in Table H-2.

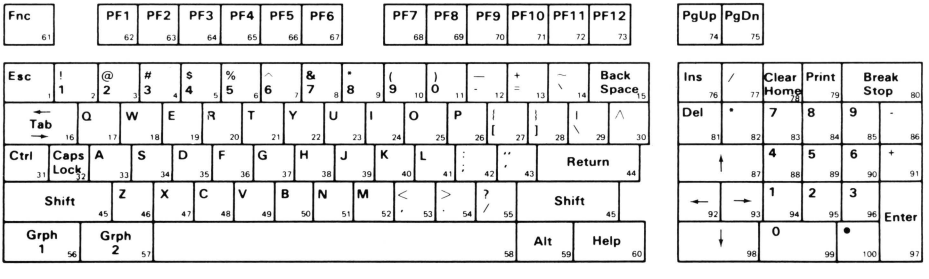


Figure H-1 APC III Keyboard Layout

Table H-2 shows the extended keyboard character set.

D7	D6	D5	D4	D3	D2	D1	D0		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
				0	0	0	0	0	1	17	35	52	83		80	45								
				0	0	0	1	1	2	18	36	53	84		79	32								
				0	0	1	0	2	3	19	37	54	85		62	56								
				0	0	1	1	3	4	20	38	55	86		63	57								
				0	1	0	0	4	5	21	39	58	88		64	31								
				0	1	0	1	5	6	22	40	74	89		65	61								
				0	1	1	0	6	7	23	41	75	90		66									
				0	1	1	1	7	8	24	42	76	91		67									
				1	0	0	0	8	9	25	43	81	94		68									
				1	0	0	1	9	10	26	44	87	95		69									
				1	0	1	0	A	11	27	46	72	96		70									
				1	0	1	1	B	12	28	46	73	97		71									
				1	1	0	0	C	13	29	48	98	99		72									
				1	1	0	1	D	14	30	49	78	100		73									
				1	1	1	0	E	15	33	50	77			59									
				1	1	1	1	F	16	34	51	82			60									

Figure H-2 Keyboard Scan Codes/Key Position Conversion Chart

Table H-1 Keyboard Character Set

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
1	ESC (1B)	ESC (1B)	ESC (1B)	No Code	No Code	No Code	00
2	1 (31)	! (21)	No Code	(00 50)	No Code	No Code	01
3	2 (32)	@ (40)	NUL (00 00)	(00 51)	No Code	No Code	02
4	3 (33)	# (23)	No Code	(00 52)	No Code	No Code	03
5	4 (34)	$ (24)	No Code	(00 53)	No Code	No Code	04
6	5 (35)	% (25)	No Code	(00 54)	No Code	No Code	05
7	6 (36)	^ (5E)	No Code	(00 55)	No Code	No Code	06
8	7 (37)	& (26)	No Code	(00 56)	No Code	No Code	07
9	8 (38)	* (2A)	No Code	(00 57)	No Code	No Code	08
10	9 (39)	((28)	No Code	(00 58)	No Code	No Code	09

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
11	0 (30)	) (29)	No Code	(00 59)	No Code	No Code	0A
12	— (2D)	— (5F)	US (1F)	(00 5A)	No Code	No Code	0B
13	= (3D)	+ (2B)	No Code	(00 5B)	No Code	No Code	0C
14	` (60)	~ (7E)	No Code	No Code	No Code	No Code	0D
15	BS (08)	BS (08)	DEL (7F)	No Code	No Code	No Code	0E
16	HT (09)	HT (09)	No Code	No Code	No Code	No Code	0F
17	q (71)	Q (51)	DC1 (11)	(00 10)	(80) (A0)	(C0) (E0)	10
18	w (77)	W (57)	ETB (17)	(00 11)	(81) (A1)	(C1) (E1)	11
19	e (65)	E (45)	ENQ (05)	(00 12)	(82) (A2)	(C2) (E2)	12
20	r (72)	R (52)	DC2 (12)	(00 13)	(83) (A3)	(C3) (E3)	13

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
21	t (74)	T (54)	DC4 (14)	(00 14)	(84) (A4)	(C4) (E4)	14
22	y (79)	Y (59)	EM (19)	(00 15)	(85) (A5)	(C5) (E5)	15
23	u (75)	U (55)	NAK (15)	(00 16)	(86) (A6)	(C6) (E6)	16
24	i (69)	I (49)	HT (09)	(00 17)	(87) (A7)	(C7) (E7)	17
25	o (6F)	O (4F)	SI (0F)	(00 18)	(88) (A8)	(C8) (E8)	18
26	p (70)	P (50)	DLE (10)	(00 19)	(89) (A9)	(C9) (E9)	19
27	[(5B)	{ (7B)	ESC (1B)	No Code	(8A) (AA)	(CA) (EA)	1A
28	] (5D)	} (7D)	GS (1D)	No Code	(8B) (AB)	(CB) (EB)	1B
29	\ (5C)	| (7C)	FS (1C)	No Code	(8C) (AC)	(CC) (EC)	1C
30	No Code	^ (5E)	RS (1E)	No Code	(8D) (AD)	(CD) (ED)	1D

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
31 Ctrl	No Code	No Code	No Code	No Code	No Code		74
32 Caps Lock	No Code	No Code	No Code	No Code	No Code		74
33	a (61)	A (41)	SOH (01)	(00 1E)	(8E) (AE)	(CE) (EE)	1E
34	s (73)	S (53)	DC3 (13)	(00 1F)	(8F) (AF)	(CF) (EF)	1F
35	d (64)	D (44)	EOT (04)	(00 20)	(90) (B0)	(D0) (F0)	20
36	f (66)	F (46)	ACK (06)	(00 21)	(91) (B1)	(D1) (F1)	21
37	g (67)	G (47)	BEL (07)	(00 22)	(92) (B2)	(D2) (F2)	22
38	h (68)	H (48)	BS (08)	(00 23)	(93) (B3)	(D3) (F3)	23
39	j (6A)	J (4A)	LF (0A)	(00 24)	(94) (B4)	(D4) (F4)	24
40	k (6B)	K (4B)	VT (0B)	(00 25)	(95) (B5)	(D5) (F5)	25

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
41	1 (6C)	L (4C)	FF (0C)	(00 26)	(96) (B6)	(D6) (F6)	26
42	; (3B)	: (3A)	No Code	No Code	(97) (B7)	(D7) (F7)	27
43	' (27)	" (22)	No Code	No Code	(98) (B8)	(D8) (F8)	28
44	CR (0D)	CR (0D)	LF (0A)	No Code	No Code	No Code	29
45 Shift	No Code	No Code	No Code	No Code	No Code	No Code	70
46	z (7A)	Z (5A)	SUB (1A)	(00 2A)	(99) (B9)	(D9) (F9)	2A
47	x (78)	X (58)	CAN (18)	(00 2B)	(9A) (BA)	(DA) (FA)	2B
48	c (63)	C (43)	ETX (03)	(00 2C)	(9B) (BB)	(DB) (FB)	2C
49	v (76)	V (56)	SYN (16)	(00 2D)	(9C) (BC)	(DC) (FC)	2D
50	b (62)	B (42)	STX (02)	(00 2E)	(9D) (BD)	(DD) (FD)	2E

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
51	n (6E)	N (4E)	SO (0E)	(00 2F)	(9E) (BE)	(DE) (FE)	2F
52	m (6D)	M (4D)	CR (0D)	(00 30)	(9F) (BF)	(DF) (FF)	30
53	, (2C)	< (3C)	No Code	No Code	No Code	No Code -	31 -
54	. (2E)	> (3E)	No Code	No Code	No Code	No Code	32
55	/ (2F)	? (3F)	No Code	No Code	No Code	No Code	33
Grph 1 56	No Code	No Code	No Code	No Code	No Code	No Code	72
Grph 2 57	No Code	No Code	No Code	No Code	No Code	No Code	73
58	SP (20)	SP (20)	SP (20)	SP (20)	No Code	No Code	34
Alt 59	No Code	No Code	No Code	No Code	No Code	No Code	6E
Help 60	HELP (00 6F)	No Code	(00 BB)	(00 CB)	No Code	No Code	6F

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
FNC 61	No Code	No Code	No Code	No Code	No Code	No Code	75
PF1 62	(00 62)	(00 8C)	(00 98)	(00 A4)	No Code	No Code	62
PF2 63	(00 63)	(00 8D)	(00 99)	(00 A5)	No Code	No Code	63
PF3 64	(00 64)	(00 8E)	(00 9A)	(00 A6)	No Code	No Code	64
PF4 65	(00 65)	(00 8F)	(00 9B)	(00 A7)	No Code	No Code	65
PF5 66	(00 66)	(00 90)	(00 9C)	(00 A8)	No Code	No Code	66
PF6 67	(00 67)	(00 91)	(00 9D)	(00 A9)	No Code	No Code	67
PF7 68	(00 68)	(00 92)	(00 9E)	(00 AA)	No Code	No Code	68
PF8 69	(00 69)	(00 93)	(00 9F)	(00 AB)	No Code	No Code	69
PF9 70	(00 6A)	(00 94)	(00 A0)	(00 AC)	No Code	No Code	6A

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
PF10 71	(00 6B)	(00 95)	(00 A1)	(00 AD)	No Code	No Code	6B
PF11 72	(00 6C)	(00 96)	(00 A2)	(00 AE)	No Code	No Code	6C
PF12 73	(00 6D)	(00 97)	(00 A3)	(00 AF)	No Code	No Code	6D
PgUp 74	Page Up (00 35)	Page Up (00 35)	Top of Text and Home (00 B0)	(00 C0)	No Code	No Code	35
PgDn 75	Page Down (00 36)	Page Down (00 36)	Erase to EOS (00 B1)	(00 C1)	No Code	No Code	36
Ins 76	Ins (00 37)	Ins (00 37)	(00 B2)	(00 C2)	No Code	No Code	37
77	/ (2F)	/ (2F)	No Code	No Code	No Code	No Code	3E
Home 78	Home (00 3D)	Home (00 3D)	Clr Scrn (00 B8)	(00 C8)	No Code	No Code	3D
Print 79	Print (00 61)	Hrd Copy (00 60)	Echo (00 BA)	(00 CA)	No Code	No Code	61

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
Break	Break	Break	Break		No Code	No Code	60
80	(00 60)	No Code	Process	(00 C9)			
81	DEL	DEL			No Code	No Code	38
Del	(00 38)	(00 38)	(00 B3)	(00 C3)			
82	*	*	No Code	No Code	No Code	No Code	3F
	(2A)	(2A)					
83	7	7	No Code	No Code	No Code	No Code	40
	(37)	(37)					
84	8	8	No Code	No Code	No Code	No Code	41
	(38)	(38)					
85	9	9	No Code	No Code	No Code	No Code	42
	(39)	(39)					
86	-	-	No Code	No Code	No Code	No Code	43
	(2D)	(2D)					
87					No Code	No Code	39
	(00 39)	(00 39)	(00 B4)	(00 C4)			
88	4	4	No Code	No Code	No Code	No Code	44
	(34)	(34)					
89	5	5	No Code	No Code	No Code	No Code	45
	(35)	(35)					

Table H-1 Keyboard Character Set (cont'd)

Keys							
Key Position	**Unshifted**	**Shift**	**Ctrl**	**Alt**	**Grph 1**	**Grph 2**	**Scan Code (hex)**
					Base Upper	Base Upper	
90	6 (36)	6 (36)	No Code	No Code	No Code	No Code	46
91	+ (2B)	(2B)	No Code	No Code	No Code	No Code	47
92 ←	← (00 3A)	← (00 3A)	Rev. Wrd (00 B5)	(00 C5)	No Code	No Code	3A
93 →	→ (00 3B)	→ (00 3B)	Adv. Wrd (00 B6)	(00 C6)	No Code	No Code	3B
94	1 (31)	1 (31)	No Code	No Code	No Code	No Code	48
95	2 (32)	2 (32)	No Code	No Code	No Code	No Code	49
96	3 (33)	3 (33)	No Code	No Code	No Code	No Code	4A
97 Enter	CR (0D)	CR (0D)	LF (0A)	No Code	No Code	No Code	4B
98	(00 3C)	(00 3C)	(00 B7)	(00 C7)	No Code	No Code	3C
99	0 (30)	0 (30)	No Code	No Code	No Code	No Code	4C
100	(2E)	(2E)	No Code	No Code	No Code	No Code	4D

Table H-2 Extended Keyboard Character Set

Second Code (Hex)	Function
02	NUL Character *Note* By default, the MS-DOS keyboard device driver will *not* return these codes, unless they have been remapped with the Re-map Key Code Escape sequence.
	←
OF	Tab
10 - 19	Alt Q, W, E, R, T, Y, U, I, O, P
1E - 26	Alt A, S, D, F, G, H, J, K, L
24 - 30	Alt Z, X, C, V, B, N, M
35	Pg Up
36	Pg Dn
37	Ins
38	Del
39 - 3C	↑, ←, →, ↓
3D	Clear Home
50 - 5B	Alt 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =, (Key 2 - 13)
60	Break Stop
61	Print
62 - 6D	PF1 - PF12 (Unshifted Function Keys)
6F	Help
80 - 8B	PF1 - PF2 (Fnc, PF1 - PF12)
8C - 97	PF1 - PF12 (Shift, PF1 - PF12)
98 - A3	PF1 - PF12 (Ctrl, PF1 - PF12)
A4 - AF	PF1 - PF12 (Alt, PF1 - PF12)

Table H-2 Extended Keyboard Character Set (cont'd)

Second Code (Hex)	Function
B0	Ctrl, Pg Up
B1	Ctrl, Pg Dn
B2	Ctrl, Ins
B3	Ctrl, Del
B4	Ctrl, ↑
B5	Ctrl, ←
B6	Ctrl, →
B7	Ctrl, ↓
B8	Ctrl, Clear Home
BA	Ctrl, Print (Start/Stop Echo to Printer)
BB	Ctrl, Help
C0	Alt, Pg Up
C1	Alt, Pg Dn
C2	Alt, Ins
C3	Alt, Del
C4	Alt, ↑
C5	Alt, ←
C6	Alt, →
C7	Alt, ↓
C8	Alt, Clear Home
C9	Alt, Break Stop
CA	Alt, Print
CB	Alt, Help

Table H-2 Extended Keyboard Character Set (cont'd)

Second Code (Hex)	Function
D0	Fnc, Pg Up
D1	Fnc, Pg Dn
D2	Fnc, Ins
D3	Fnc, Del
D4	Fnc, ↑
D5	Fnc, ←
D6	Fnc, →
D7	Fnc, ↓
D8	Fnc, Clear Home
D9	Fnc, Break Stop
DA	Fnc, Print
DB	Fnc, Help

Table H-3 outlines the guidelines for key usage when performing commonly used functions.

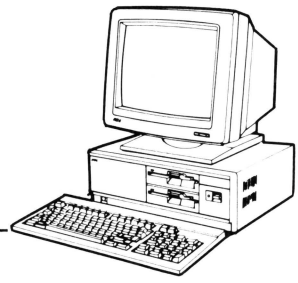
Table H-3 Suggested Keyboard Functions

Keys	Function	Comments
Home	Home cursor	Editors, word processing
Home	Return to outermost menu	Menu-driven applications
↑	Move cursor up	Full screen editor, word processing
Pg Up	Page up, scroll	Editors, word processing, backwards 25 lines, and home
←	Move cursor left	Text command entry
→	Move cursor right	Text command entry
↓	Move cursor down	Full screen editor, word processing
Pg Dn	Page down, scroll	Editors, word processing, forwards 25 lines, and home
Ins	Start/Stop insert text at cursor. Shift text right in buffer.	Text command entry
Del	Delete character at cursor	Text command entry
Back Space	Destructive back space	Text command entry
Tab →	Tab forward	Text entry

Table H-3 Suggested Keyboard Functions (cont'd)

Keys	Function	Comments
Tab ←	Tab backward	Text entry
Ctrl, Clear Home	Clear screen and home	Command entry
Esc	Exit/escape	Editor, 1 level of menu
Ctrl →	Advance word	Text entry
Ctrl ←	Reverse word	Text entry
Break Stop	Suspend system (pause)	Stop list, stop program, etc. Resumes on any key.
Ctrl, Break Stop	Break Interrupt	Interrupts current process
Fnc, Ctrl, Break Stop	System Reset	Reboots the system
Ctrl, Pg Up	Top of document and home cursor	Editors, word processing
PF1-PF12	Standard function keys	Primary function keys
Fnc, Shift, Ctrl, or Alt, PF1-PF12	Secondary function keys	Extra function keys
Alt 2-13	Extra function keys	
Alt A-Z	Extra function keys	Used when function starts with same letter as one of the alpha keys.

Appendix I
APC III Hard Disk Information



This appendix provides technical information regarding the APC III hard disk options. Information on setting up and operating the hard disk under MS-DOS is presented in the *APC III Supplement to the MS-DOS User's Guide*. A detailed description of the hard disk controller may be found in the *APC III System Reference Guide*.

In order to provide the flexibility of logically segmenting the hard disk storage area for different operating systems, users, and specialized storage requirements, NEC has implemented a partitioning scheme that is outlined in this appendix. The master hard disk boot record (stored in the first sector of the hard disk) and the Hard Disk Partition Table (stored in the second sector of the hard disk) provide the means of accessing any partition easily.

Hard Disk Partition Table

The Hard Disk Partition Table, described in Table I-1, contains all information concerning the hard disk partitions. It resides on the second sector of track 0. Under MS-DOS, this table is created (for an uninitialized hard disk) and maintained by the HDSETUP utility.

Table I-1 The Hard Disk Partition Table

Offset	Length	Parameter
+00H	1 byte	Number of partitions
+01H	1 byte	Current hard disk boot partition number
+02H	30 bytes	Reserved
+20H	32 bytes	Partition #1 parameter block
+40H	32 bytes	Partition #2 parameter block
.	.	.
.	.	.
.	.	.
+180H	32 bytes	Partition #12 parameter block
+1A0H	92 bytes	Reserved
+1FCH	2 bytes	Reserved (must = 0 for this release of MS-DOS)
+1FEH	2 bytes	ID (must = 55AAH)

The ID bytes must be present to signify a valid partition table. They also appear at the same offset in the master hard disk boot record and in the boot record of all MS-DOS partitions. If these values do not appear when expected, the record is considered invalid.

Each 32-byte partition parameter block consists of the following:

- 1 byte — Active partition indicator (1 = active, 0 = not)
- 1 byte — Operating system indicator (1 = MS-DOS)
- 2 bytes — Reserved
- 1 Dword — Location of initial program loader (sector number)
- 1 Dword — Partition start sector number
- 1 Dword — Partition end sector number
- 10 bytes — Partition name
- 6 bytes — Reserved

The double word sector numbers indicating the partition start, end, and IPL location are given relative to the very beginning of the hard disk, starting at sector 0. For example, if partition #1 begins on sector 34, the partition begins on sector 35 of the hard disk. This sector will become "relative" sector 0 within the partition itself.

MS-DOS Partition Format

Hard disks are formatted with 17 sectors (512 bytes per sector) per track. Partitions must be allocated on track boundaries, with the first 2 tracks of the hard disk reserved. File allocation unit size depends on the size of the partition. If the partition is 620 tracks or less (up to 5 megabytes), it will be formatted with 4 sectors per allocation unit (2048 bytes). If the partition is between 621 and 1240 tracks (up to 10 megabytes), it will be formatted with 8 sectors per allocation unit (4096 bytes). The first sector of the partition is reserved for the initial program loader, after which two FATs are allocated with a length of 8 sectors each. The root directory can have up to 272 entries.

System Initialization

Following a power-on or reset, MS-DOS is booted and initialized in the following way.

1. The battery backed non-volatile memory is read for the system boot drive indicator. If this value designates a hard disk drive as the boot drive, the master hard disk boot record (located on sector 0) of that drive is read.
2. The master hard disk boot record contains code which reads the Hard Disk Partition Table. The second byte of the partition table contains the number of the boot partition. This value is set either by the BOOT utility or by selecting a boot volume by holding down the corresponding PF key at boot time. (The hard disk boot process is described in the *APC III Supplement to the MS-DOS User's Guide*.)

3. If no partition has been designated or no PF key was held, the system will attempt to boot from a diskette drive. This cycle will continue until a system diskette is placed in the diskette drive or the system is powered down.
4. If the Hard Disk Partition Table has a designated boot partition, the code from the master hard disk boot record will then read the parameter block of that partition for two important indicators. First, it will confirm that it is an active partition.
5. Second, it will read the sector containing the initial program loader in that partition. If the ID bytes (55AAH) are found at the end of the IPL sector, then program control is given to the IPL.
6. The IPL will search the directory of the partition for the system files named IO.SYS and MSDOS.SYS and then begin loading them.
7. If the system files do not exist, the system will try to boot from a diskette drive. It is necessary to use a system diskette in drive A: or B: if the active hard disk partition has not been formatted with the /S option.

Sample Hard Disk

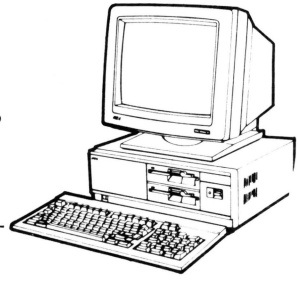
Table I-2 outlines the contents of a sample hard disk by sector number.

Table I-2 Sample Hard Disk

Sector Number	Contents
Sector 0	Master hard disk boot record
Sector 1	Hard Disk Partition Table
	Offset **Value** **Description**
	+00H 04 Number of partitions.
	+01H 02 Boot volume is partition #2.
	. . .
	+40H 01 First byte of partition #2 parameter block; it is active.
	+44H 27FAH Location of IPL.
Sector 27FAH	IPL code
Sector 27FBH Through 280BH	FATs for partition #2
Sector 2800CH	Root Directory with IO.SYS and MSDOS.SYS

Appendix J

GRAPH Installable Driver (GRAPHIC.SYS)



The GRAPH Installable driver, written by NEC, provides a universal and easy-to-use method of accessing the vast graphic capabilities of the APC III. The driver consists of a character output interface that may be used by most MS-DOS 2.11 programming languages, utilities, and applications that are capable of sending output to a character oriented device.

This appendix describes the installation of the driver and the purpose and syntax of its commands and operations. It also includes a sample M-BASIC program to demonstrate the use of the driver.

Applications

In many cases, a user or programmer finds it necessary to write a specialized and non-generic program that interfaces an application or utility to the graphics hardware of the APC III. GSX does not provide an easy-to-understand interface to graphics hardware. The GRAPH Installable driver provides a character oriented interface that is easy to use and understand. Only a limited vocabulary of commands need to be learned in order to take advantage of the APC III graphics hardware.

How to Use the GRAPH Installable Driver

The following steps describe how to install the GRAPH Installable driver on your system so it is available for use.

1. Create or modify the CONFIG.SYS file, which is located in the root directory of the diskette or hard disk volume to be booted, to include the following line of text.

   DEVICE = GRAPHIC.SYS <cr>

2. Copy the GRAPHIC.SYS file into the root directory of the bootable disk volume or system diskette.
3. Reboot the APC III via the <Fnc> + <Ctrl> + <Break/Stop> key sequence, RESET switch, or powering on. Be sure to boot from the system diskette or hard disk volume that contains the CONFIG.SYS and GRAPHIC.SYS files.
4. Use the GRAPH Installable driver as a character I/O device with the name of "GRAPH".

Refer to the section on Operation and Command Descriptions for a complete list of the capabilities for the "GRAPH" device.

Syntax and Data Conventions

The following notes apply to the syntax and use of the GRAPHIC.SYS installable driver. Read these notes carefully to understand the general syntax of the driver's commands.

1. The bracket characters ([and]) enclose optional parameters. The angle bracket characters (< and >) enclose required parameters. The underscore character (_) denotes that a space is required to separate the option and the attribute fields of a parameter.
2. The comma and semicolon characters (, and ;) are significant and must be used within the syntax.
3. The vertical bar character (|) indicates an exclusive OR.
4. The values for x1 and x2 must be in the range of 0-639. A command that exceeds this range is ignored and the program continues.

5. The values for y1 and y2 must be in the range of 0-399. A command that exceeds this range is ignored and the program continues.

6. The values for radius should only be in the range 0-199. Although the radius can be specified within a range of 0-319, values larger than 199 can result in memory area overwrites in adjacent color planes.

7. When specifying the arc begin and arc end values for ARC drawing, the following rules apply.

   - When it is desirable to draw arcs that start at angles less than 360 degrees and extend beyond 360 degrees, then the ending angle must have a value within the range of 360-720 degrees.
   - Angle values are in increments of 1 degree; they must be integer values.

8. RECT, MOVE, COPY, and ERASE operations must have the x1 and y1 values specified as the upper left corner and the x2 and y2 values as the lower right corner for the rectangle to be drawn. The x3 and y3 values for the COPY and MOVE operations define the new upper left corner for the specified block's destination.

9. The x1 and y1 values for CIR and ARC drawing are used as the center.

10. MOUSE and JOYSTICK are mutually exclusive selections but may be changed dynamically.

11. The color option <cnum> for the FILL action represents the current border color for the area to be filled. Any color that is not equal to the specified border color will be overdrawn in the area to be filled. Attempting to fill an area that has no discernable borders will cause the fill to escape to other areas on the screen. Borders are very important to the filling process, and areas for polygons (including circles, rectangles, etc.) must have solid borders (line style 0 or 7) to ensure a proper fill operation.

12. Text strings are limited to 80 characters each. Text strings exceeding 80 characters will be truncated after the 80th character.

13. The x1 and y1 values for the TEXT action specify the upper left corner of the first character's 8 x 16 or 8 x 8 block.

14. In order to read the MOUSE or JOYSTICKs, a program or utility must simply perform a read operation from the GRAPH device. The following describes the structure of the data that is returned upon the read request.

Structure of MOUSE Data

- Current X coordinate of 000-639 in ASCII decimal followed by either a space or a comma (refer to the SET operation for more detail). Three characters/digits are always returned with leading zeroes.
- Current Y coordinate of 000-399 in ASCII decimal followed by either a space or a comma (refer to the SET operation for more detail). Three characters/digits are always returned with leading zeroes.
- The most recent switch code in ASCII decimal as follows:

  0 = No switch pressed
  1 = Switch 1 pressed (left)
  2 = Switch 2 pressed (middle)
  4 = Switch 3 pressed (right)

- Terminator code of <cr> ends the sequence.

Structure of JOYSTICK Data

- Current X coordinate of 000-639 for JOYSTICK A in ASCII decimal followed by either a space or a comma (refer to the SET operation for more detail). Three characters/digits are always returned with leading zeroes.
- Current Y coordinate of 000-399 for JOYSTICK A in ASCII decimal followed by either a space or a comma (refer to the SET operation for more detail). Three characters/digits are always returned with leading zeroes.
- Current X coordinate of 000-639 for JOYSTICK B in ASCII decimal followed by either a space or a comma (refer to the SET operation for more detail). Three characters/digits are always returned with leading zeroes.
- Current Y coordinate of 000-399 for JOYSTICK B in ASCII decimal followed by either a space or a comma (refer to the SET operation for more detail). Three characters/digits are always returned with leading zeros.

- The most recent switch code in ASCII decimal as follows:

  0 = No switch pressed
  1 = Switch 1 pressed for JOYSTICK A
  2 = Switch 2 pressed for JOYSTICK A
  4 = Switch 1 pressed for JOYSTICK B
  8 = Switch 2 pressed for JOYSTICK B

- Terminator code of <cr> ends the sequence.

15. The INIT operation sets the coordinate values to the upper left (000,000) position of the display screen and the switch code to 0 for MOUSE. JOYSTICK coordinates are set to 320,200 and the switch codes are set to 0.

16. All optional parameters, when specified, supersede the corresponding default parameters set by the SET command.

Operation and Command Descriptions

This section describes the syntax and purpose of the operations and commands provided by the GRAPH Installable driver.

ARC,<x1>,<y1>,<radius>,<arc begin>,<arc end>,
<cnum>,[<line style>],[<screen spec>]<cr>

Draws an arc with a center at the coordinates specified by <x1> and <y1> having a radius specified by <radius>. The angle that is subtended by the arc is specified by the <arc begin> and <arc end> parameters (refer to Syntax and Data Conventions, Note 7, for more details). The color of the border line is specified by the <cnum> parameter. The border line will be drawn with the line style specified by the current default value set by the SET command or the optional <line style> parameter. The circle will be drawn in the default screen set by the SET command or in the screen specified by the optional <screen spec> parameter.

CIR,<x1>,<y1>,<radius>,<cnum>,[<line style>],
[<screen spec>]<cr>

Draws a circle with a center at the coordinates specified by <x1> and <y1> having a radius specified by <radius>. The color of the border line is specified by the <cnum> parameter. The border line will be drawn with the line style specified by the current default value set by the SET command or the optional <line style> parameter. The circle will be drawn in the default screen set by the SET command or in the screen specified by the optional <screen spec> parameter.

CLEAR<cr>

Clears the entire default screen specified by the SET command.

COPY,<x1>,<y1>,<x2>,<y2>,<x3>,<y3><cr>

Copies the area enclosed by the rectangle defined by the coordinates specified by the <x1>, <y1>, <x2>, <y2> parameters to an equal-sized rectangular area with its upper left corner specified by the <x3> and <y3> parameters. The upper left corner for the area to be moved is defined by the <x1> and <y1> parameters. No orientation changes are performed during the copy operation.

CURSOR,<ON|OFF><cr>

Activates and deactivates the display of cursors that follow the movement of the MOUSE and JOYSTICKs. The cursor color for JOYSTICKs A and B are red and green respectively. The cursor color for MOUSE is white.

ERASE,<x1>,<y1>,<x2>,<y2><cr>

Fills the area that is enclosed by the rectangle defined by the <x1>, <y1>, <x2> and <y2> parameters with the color 0 (black).

FILL,<x1>,<y1>,<cnum>,<fill spec><cr>

Fills an area bounded by a color specified by the <cnum> parameter. The new color and pattern are specified by the <fill spec> parameter. The starting coordinate for the fill operation, commonly called the "seed," is specified by the <x1> and <y1> parameters.

INIT<cr>

The INIT command must be executed once before any other commands may be sent to the GRAPH device. The INIT command presets the following conditions.

Mode	= color or monochrome determined by system hardware
Resolution	= 640 x 400
Display screen	= 0
Drawing screen	= 0

Cursor addresses and switch codes are set to logical zero. (Refer to Syntax and Data Conventions, Note 15, for more details.)

Palette registers are initialized as follows.

Register Number	**Value**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

JOYSTICK,<ON | OFF><cr>

Activates the JOYSTICKs interface and mutually excludes the use of the MOUSE.

LINE,<x1>,<y1>,<x2>,<y2>,<cnum>,[<line style>], [<screen spec>]<cr>

Draws a line between the coordinates specified by the <x1>, <y1> and <x2>, <y2> parameters. The color of the line is specified by the <cnum> parameter. The line will be drawn with the line style specified by the current default value set by the SET command or the optional <line style> parameter. The line will be drawn in the default screen set by the SET command or in the screen specified by the optional <screen spec> parameter.

MOUSE,<ON | OFF><cr>

Activates and selects the type of MOUSE to be used on the standard serial port (AUX) of the APC III. This command also mutually excludes the use of the JOYSTICKs.

MOVE,<x1>,<y1>,<x2>,<y2>,<x3>,<y3><cr>

Copies the area enclosed by the rectangle defined by the coordinates specified by the <x1>, <y1>, <x2>, <y2> parameters to an equal-sized rectangular area with its upper left corner specified by the <x3> and <y3> parameters. The upper left corner for the area to be moved is defined by the <x1> and <y1> parameters. No orientation changes are performed during the copy operation. After the copy is performed, then the area that is enclosed by the rectangle defined by the <x1>, <y1>, <x2> and <y2> parameters will be filled with the color 0 (black).

RECT,<x1>,<y1>,<x2>,<y2>,<cnum>,[<line style>], [<screen spec>]<cr>

Draws a rectangle with a height of <x2> minus <x1> and a width of <y2> minus <y1> starting at the upper left corner specified by the <x1> and <y1> parameters. The color of the border line is specified by the <cnum> parameter. The border line will be drawn with the line style specified by the current default value set by the SET command or the optional <line style> parameter. The rectangle will be drawn in the default screen set by the SET command or in the screen specified by the optional <screen spec> parameter.

SET,<<line style>|<screen spec>|<font spec>
|<pallette spec>|<mouse spec>|<sep spec>><cr>

Changes the value of the default parameters for the GRAPH device. (Refer to the Parameter Syntax Specifications for more detail.)

TEXT,<x1>,<y1>,<cnum>,<text string>,[<font spec>],
[<screen spec>]<cr>

Draws up to 80 characters of text in the default font style specified by the SET command or in the font style specified by the optional <font spec> parameter. The text will be drawn starting at the coordinates specified by the <x1> and <y1> parameters. The starting coordinate defines the upper left corner of the first character's 8 x 16 or 8 x 8 block. The color of the text is specified by the <cnum> parameter. The text will be drawn in the default screen set by the SET command or in the screen specified by the optional <screen spec> parameter.

Parameter Syntax Specifications

line style = LINE_<number range>

```
0 = ********** invisible **********
1 = * * * * * * * * * * * * * * * * * * * *
2 = **  **  **  **  **  **  **  **  **  **
3 =  *** *** *** *** *** *** *** ***
4 = ****    ****    ****    ****    ****
5 = ********        ********        ********
6 = ******  ******  ******  ******  ******
7 = ****************************************
```

Line style 0 is solid but invisible.

text string = <alpha|digit>[<alpha|digit>]...

fill spec = FILL_<cnum>;<pnum>

screen spec = SCREEN_<0|1>

font spec = FONT_<0|1>

0 = 8 x 16 font (normal APC III)
1 = 8 x 8 font (reduced APC III)

pallette spec = PALLETTE_<number sequence>

mouse spec = MOUSE_<0 | 1>

0 = Mouse Systems Mouse
1 = SummaGraphics Mouse

sep spec = SEP_<,|_>

The separator specification determines whether a comma or a space is inserted between groups of digits returned from the MOUSE or JOYSTICK.

cnum = <number range>

0 = black
1 = green
2 = red
3 = yellow
4 = blue
5 = cyan
6 = magenta
7 = white

pattern = PATTERN_<number range>

0 = half intensity
1 = solid fill
2 = vertical lines
3 = horizontal lines
4 = –45 degree diagonals
5 = +45 degree diagonals
6 = "X" crosshatch
7 = right-angle crosshatch

See Figure J-1 for pattern illustration.

pnum = <number range>

Use numbers corresponding to the same patterns as shown for the pattern parameter.

number
sequence = <<number range>;<number range>;<number range>;
<number range>;<number range>;<number range>;
<number range>;<number range>>

number range = <0 | 1 | 2 | 3 | 4 | 5 | 6 | 7>

alpha = < A | B | C | D | E | F | G | H | I | J | K | L |
M | N | O | P | Q | R | S | T | U | V | W | X |
Y | Z | , | . | ' | / | < | > | ? | : | " | ; |
[|] | { | } | \ | ^ | + | = | _ | - | ! | @ |
| $ | % | ^ | & | * | (|) | ~ | ` | a | b |
c | d | e | f | g | h | i | j | k | l | m | n |
o | p | q | r | s | t | u | v | w | x | y | z >

digit = <0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9>

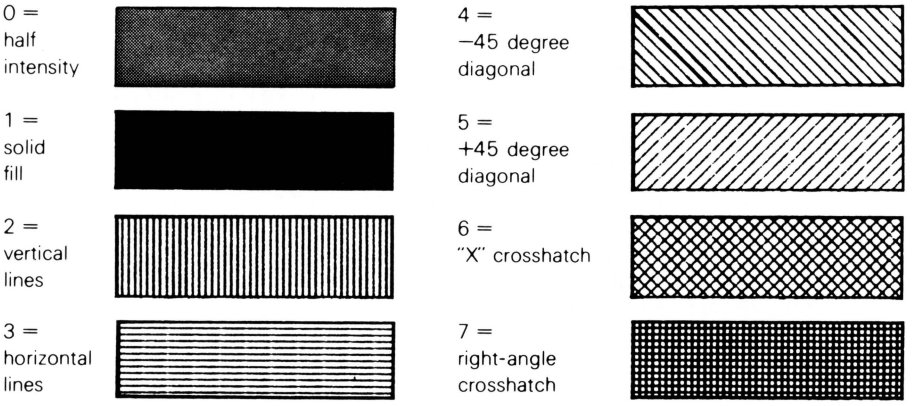


Figure J-1 Available Patterns

Sample Program

The program listing shown in Figure J-2 is a simple M-BASIC program that produces output that is directed to the GRAPH device. The output from the program, shown in Figure J-3, produces the graphics output shown in Figure J-4.

Although the program directs its output to the GRAPH device, the output could be directed to a regular disk file instead of the GRAPH device, and then the disk file or an edited version could be copied to the GRAPH device (COPY file to GRAPH).

```
10 OPEN "O",#1,"GRAPH"
20 PRINT CHR$(27);"[2J";
30 PRINT #1,"INIT"
40 PRINT #1,"CLEAR"
50 X1=0:Y1=0:Y2=200
60 FOR X2=9 TO 639 STEP 70
65 C=(C+1) AND 7
70 PRINT #1,"RECT,";X1;",";Y1;",";X2;",";Y2;",";C
90 NEXT X2
100 FOR R=0 TO 199 STEP 40
105 C=(C+1) AND 7
110 PRINT #1,"CIR,320,200,";R;",";C
130 NEXT R
140 FOR R=0 TO 200 STEP 40
145 C=(C+1) AND 7
150 PRINT #1,"ARC,0,0,";R;",270,360,";C
170 PRINT #1,"ARC,639,0,";R;",180,270,";C
190 NEXT R
200 FOR Y1=0 TO 399 STEP 40
205 C= (C+1) AND 7
210 PRINT #1,"LINE,0,";Y1;",320,200,";C
220 PRINT #1,"LINE,639,";Y1;",320,200,";C
230 NEXT Y1
240 CLOSE
250 END
```

Figure J-2 Sample M-BASIC Program for Using the GRAPH Device

```
INIT
CLEAR
RECT, 0 , 0 , 9 , 200 , 1
RECT, 0 , 0 , 79 , 200 , 2
RECT, 0 , 0 , 149 , 200 , 3
RECT, 0 , 0 , 219 , 200 , 4
RECT, 0 , 0 , 289 , 200 , 5
RECT, 0 , 0 , 359 , 200 , 6
RECT, 0 , 0 , 429 , 200 , 7
RECT, 0 , 0 , 499 , 200 , 0
RECT, 0 , 0 , 569 , 200 , 1
RECT, 0 , 0 , 639 , 200 , 2
CIR,320,200, 0 , 3
CIR,320,200, 40 , 4
CIR,320,200, 80 , 5
CIR,320,200, 120 , 6
CIR,320,200, 160 , 7
ARC,0,0, 0 ,270,360, 0
ARC,639,0, 0 ,180,270, 0
ARC,0,0, 40 ,270,360, 1
ARC,639,0, 40 ,180,270, 1
ARC,0,0, 80 ,270,360, 2
ARC,639,0, 80 ,180,270, 2
ARC,0,0, 120 ,270,360, 3
ARC,639,0, 120 ,180,270, 3
ARC,0,0, 160 ,270,360, 4
ARC,639,0, 160 ,180,270, 4
ARC,0,0, 200 ,270,360, 5
ARC,639,0, 200 ,180,270, 5
LINE,0, 0 ,320,200, 6
LINE,639, 0 ,320,200, 6
LINE,0, 40 ,320,200, 7
LINE,639, 40 ,320,200, 7
LINE,0, 80 ,320,200, 0
LINE,639, 80 ,320,200, 0
LINE,0, 120 ,320,200, 1
LINE,639, 120 ,320,200, 1
LINE,0, 160 ,320,200, 2
LINE,639, 160 ,320,200, 2
LINE,0, 200 ,320,200, 3
LINE,639, 200 ,320,200, 3
LINE,0, 240 ,320,200, 4
LINE,639, 240 ,320,200, 4
LINE,0, 280 ,320,200, 5
LINE,639, 280 ,320,200, 5
LINE,0, 320 ,320,200, 6
LINE,639, 320 ,320,200, 6
LINE,0, 360 ,320,200, 7
LINE,639, 360 ,320,200, 7
```

Figure J-3 Output of M-BASIC Program

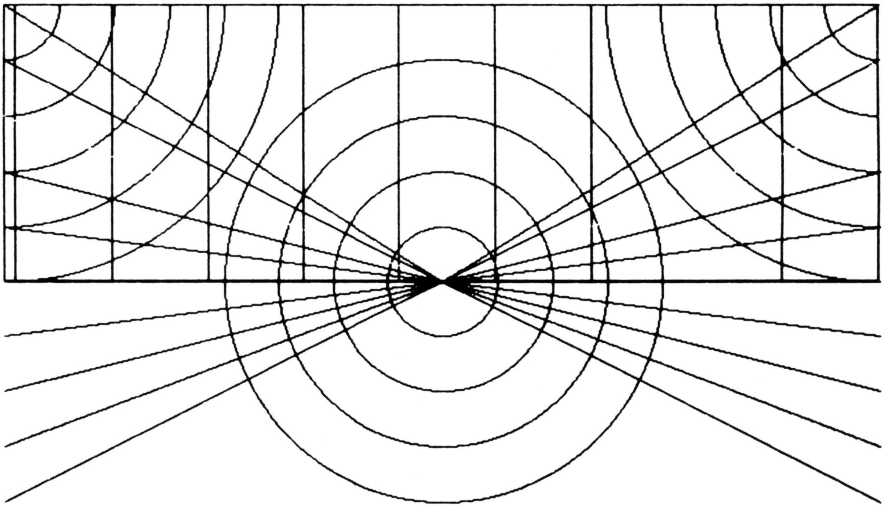


Figure J-4 Graphics Display from Output of M-BASIC Program

APC III

NEC
NEC Information Systems, Inc.

USER'S COMMENTS FORM

Document: APC III Supplement to MS-DOS Programmer's Reference Guide

Document No.: 819-150012-000 Rev. 01

Please suggest improvements to this manual.

Please list any errors in this manual. Specify by page.

From:

Name _____

Title _____

Company _____

Address _____

Dealer Name _____

Date: _____

Please cut along this line.

Seal or tape all edges for mailing - do not use staples.

FOLD HERE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 105 BOXBOROUGH, MA

POSTAGE WILL BE PAID BY ADDRESSEE

NEC Information Systems, Inc.
Dept: Publications
1414 Mass. Ave.
Boxborough, MA 01719

FOLD HERE

Seal or tape all edges for mailing - do not use staples.

Microsoft® MS™-DOS

Operating System

Programmer's Reference Manual

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the Programmer's Reference Manual on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

(C) Microsoft Corporation 1981, 1983, 1984

Portions of this manual (C) Intel Corporation 1980

Comments about this documentation may be sent to:

Microsoft Corporation
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

Microsoft is a registered trademark of Microsoft Corporation.

MS is a registered trademark of Microsoft Corporation.

XENIX is a trademark of Microsoft Corporation.

CP/M is a registered trademark of Digital Research, Inc.

INTEL is a registered trademark of Intel Corporation.

Document No. 8411-200-01
Part No. 036-014-008

APC III

NEC INFORMATION SYSTEMS PERSONAL COMPUTER
LIMITED WARRANTY
SOFTWARE REGISTRATION CARD

PLEASE READ THE FOLLOWING TEXT CAREFULLY. IT CONSTITUTES A CONTINUATION OF THE PROGRAM LICENSE AGREEMENT FOR THE SOFTWARE APPLICATION PROGRAM CONTAINED IN THIS PACKAGE.

If you agree to all the terms and conditions contained in both parts of the Program License Agreement, please fill out the detachable postcard and return it to:

NEC Information Systems, Inc.
1414 Massachusetts Ave.
Boxborough, MA 01719

Attention: Customer Technical Services

LIABILITY

In no event shall the copyright holder, the original licensor nor any intermediate sublicensors of this software be responsible for any indirect or consequential damages or lost profits arising from the use of this software.

COPYRIGHT

The name of the copyright holder of this software must be recorded exactly as it appears on the label of the original diskette as supplied by NECIS on a label attached to each additional copy you make.

You must maintain a record of the number and location of each copy of this program.

All NECIS software programs and copies remain the property of the copyright holder, though the physical medium on which they exist is the property of the licensee.

MERGING, ALTERATION

Should this program be merged with or incorporated into another program, or altered in any way by the licensee, the terms of the Warranty contained herein are voided and neither NECIS nor the copyright holder nor any intermediate sublicensors will assure the conformity of this software to its specification nor refund the license fee for such nonconformity.

Upon termination of this license for any reason, any such merged or incorporated programs must be separated from the programs with which they have been merged or incorporated and any altered programs must be destroyed.

Please fill in this form completely and carefully. Failure to do so will terminate your license agreement for this software.

When you have completed this form, detach it from the rest of the Program License Agreement, and drop in any mailbox. The remaining portion of the license agreement should be retained for your records.

819-000100-6D01 Rev. 01

"The undersigned Enduser of this NECIS software hereby acknowledges that he/she has read and fully understands the terms and conditions of this license agreement, hereby incorporated into this card and acknowledged by this reference.

The undersigned hereby acknowledges that by signing this document, he/she becomes a party to said license with NECIS, and agrees to be bound by all terms, conditions, and obligations contained therein."

PROGRAM NAME (as it appears on diskette label):

_____

SERIAL NUMBER: _____ DATE PURCHASED: _____

DEALER NAME: _____ CITY: _____ STATE: _____

YOUR NAME: _____

YOUR COMPANY (only if company purchase): _____

YOUR ADDRESS: _____

YOUR CITY, STATE, ZIP: _____

YOUR TELEPHONE: _____

"Warranty Requires Return of This Card" 819-000100-6D01 Rev. 01

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 105 BOXBOROUGH, MA

POSTAGE WILL BE PAID BY ADDRESSEE

NEC Information Systems, Inc.
1414 Massachusetts Ave.
Boxborough, MA 01719

Attention: Customer Technical Services

System Requirements

Disk drive(s)

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

For more information about other Microsoft products, contact:

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

GENERAL INTRODUCTION

The Microsoft(R) MS(tm)-DOS Programmer's Reference Manual is a technical reference manual for system programmers. This manual contains the following:

Chapter 1: a description and examples of all MS-DOS 2.0 system calls and interrupts.

Chapter 2: information on how to install your own device drivers on MS-DOS. Two examples of device driver programs (one serial and one block) are included.

Chapter 3: technical information about MS-DOS, including MS-DOS disk allocation, the File Allocation Table, and MS-DOS standard disk formats.

Chapter 4: a description of MS-DOS control blocks and work areas, including a memory map.

Chapter 5: information on .EXE file structure and loading.

Chapter 6: INTEL(R) object module formats.

Contents

General Introduction

Chapter 1 System Calls

1.1 Introduction 1-1
1.1.1 About Interrupts 1-2
1.1.2 About Function Requests 1-2
1.2 Calls Compatible with the XENIX Operating System 1-2
1.3 Using the System Calls 1-3
1.3.1 Issuing an Interrupt 1-3
1.3.2 Calling A Function Request 1-3
1.3.3 Using the Calls From a High-Level Language 1-4
1.3.4 Treatment of Registers 1-4
1.3.5 Error Checking 1-4
1.3.6 Returning Control to MS-DOS 1-6
1.4 Hierarchical Directory System 1-7
1.4.1 Handles 1-8
1.4.2 Directory Entry 1-9
1.4.3 Fields of the Directory Entry 1-9
1.4.4 File Attributes 1-10
1.5 System Call Descriptions 1-10
1.5.1 Sample Programs 1-11
1.5.2 Error Handling 1-13
1.6 File Control Block (FCB) 1-14
1.6.1 Fields of the FCB 1-14
1.6.2 Extended FCB 1-16
1.7 Interrupts 1-21
1.8 Function Requests 1-35
1.9 Macro Definitions for MS-DOS System 1-169

Chapter 2 MS-DOS 2.0 Device Drivers

2.1 What is a Device Driver? 2-1
2.2 Device Headers 2-3
2.2.1 Pointer to Next Device Field 2-3
2.2.2 Attribute Field 2-4
2.2.3 Strategy and Interrupt Routines 2-5
2.2.4 Name Field 2-5
2.3 How to Create a Device Driver 2-5
2.4 Installation of Device Drivers 2-6
2.5 Request Header 2-6
2.5.1 Unit Code 2-7
2.5.2 Command Code Field 2-7
2.5.3 MEDIA CHECK and BUILD BPB 2-8
2.5.4 Status Word 2-9
2.6 Function Call Parameters 2-11
2.6.1 INIT 2-12
2.6.2 MEDIA CHECK 2-12
2.6.3 BUILD BPB 2-13

2.6.4 Media Descriptor Byte 2-15
2.6.5 READ OR WRITE 2-16
2.6.6 NON DESTRUCTIVE READ NO WAIT 2-17
2.6.7 STATUS 2-18
2.6.8 FLUSH 2-18
2.7 The CLOCK Device 2-19
2.8 Example of Device Drivers 2-20
2.8.1 Block Device Driver 2-20
2.8.2 Character Device Driver 2-34

Chapter 3 MS-DOS Technical Information

3.1 MS-DOS Initialization 3-1
3.2 The Command Processor 3-1
3.3 MS-DOS Disk Allocation 3-2
3.4 MS-DOS Disk Directory 3-3
3.5 File Allocation Table 3-7
3.5.1 How to Use the File Allocation Table 3-8
3.6 MS-DOS Standard Disk Formats 3-9

Chapter 4 MS-DOS Control Blocks and Work Areas

4.1 Typical MS-DOS Memory Map 4-1
4.2 MS-DOS Program Segment 4-2

Chapter 5 EXE File Structure and Loading

Chapter 6 Intel Relocatable Object Module Formats

6.1 Introduction 6-1
6.2 Definition of Terms 6-2
6.3 Module Identification and Attributes 6-6
6.4 Segment Definition 6-6
6.5 Segment Addressing 6-6
6.6 Symbol Definition 6-7
6.7 Indices
6.8 Conceptual Framework for Fixups 6-8
6.9 Self-Relative Fixups 6-14
6.10 Segment-Relative Fixups 6-15
6.15 Record Order 6-15
6.12 Introduction to the Record Formats 6-18
6.13 Numeric List of Record Types 6-50
6.14 Intel Type Representations 6-51
6.15 Microsoft Type Representations for Communal Variables 6-55

Index

Contents

Chapter 1 System Calls

1.1 Introduction 1-1
1.1.1 About Interrupts 1-2
1.1.2 About Function Requests 1-2
1.2 Calls Compatible with the XENIX Operating System 1-2
1.3 Using the System Calls 1-3
1.3.1 Issuing an Interrupt 1-3
1.3.2 Calling A Function Request 1-3
1.3.3 Using the Calls From a High-Level Language 1-4
1.3.4 Treatment of Registers 1-4
1.3.5 Error Checking 1-4
1.3.6 Returning Control to MS-DOS 1-6
1.4 Hierarchical Directory System 1-7
1.4.1 Handles 1-8
1.4.2 Directory Entry 1-9
1.4.3 Fields of the Directory Entry 1-9
1.4.4 File Attributes 1-10
1.5 System Call Descriptions 1-10
1.5.1 Sample Programs 1-11
1.5.2 Error Handling 1-13
1.6 File Control Block (FCB) 1-14
1.6.1 Fields of the FCB 1-14
1.6.2 Extended FCB 1-16
1.7 Interrupts 1-21
20H Program Terminate 1-22
21H Function Request 1-24
22H Terminate Address 1-25
23H CONTROL-C Exit Address 1-25
24H Fatal Error Abort Address 1-26
25H Absolute Disk Read 1-30
26H Absolute Disk Write 1-32
27H Terminate But Stay Resident 1-34
1.8 Function Requests 1-35
00H Terminate Program 1-36
01H Read Keyboard and Echo 1-38
02H Display Character 1-39
03H Auxiliary Input 1-40
04H Auxiliary Output 1-41
05H Print Character 1-42
06H Direct Console I/O 1-44
07H Direct Console Input 1-46
08H Read Keyboard 1-47
09H Display String 1-48
0AH Buffered Keyboard Input 1-49
0BH Check Keyboard Status 1-51
0CH Flush Buffer, Read Keyboard 1-52
0DH Disk Reset 1-53
0EH Select Disk 1-54

0FH Open File 1-56
10H Close File 1-58
11H Search for First Entry 1-60
12H Search for Next Entry 1-62
13H Delete File 1-64
14H Sequential Read 1-66
15H Sequential Write 1-68
16H Create File 1-70
17H Rename File 1-72
19H Current Disk 1-74
1AH Set Disk Transfer Address 1-75
21H Random Read 1-77
22H Random Write 1-79
23H File Size 1-81
24H Set Relative Record 1-83
25H Set Vector 1-85
27H Random Block Read 1-86
28H Random Block Write 1-88
29H Parse File Name 1-90
2AH Get Date 1-93
2BH Set Date 1-95
2CH Get Time 1-97
2DH Set Time 1-98
2EH Set/Reset Verify Flag 1-100
2FH Get Disk Transfer Address 1-102
30H Get DOS Version Number 1-103
31H Terminate But Stay Resident 1-104
33H CONTROL-C Check 1-105
35H Get Vector 1-107
36H Get Disk Free Space 1-109
38H Get Country Data 1-111
39H Create Directory 1-114
3AH Remove Directory 1-116
3BH Change Current Directory 1-118
3CH Create Handle 1-120
3DH Open Handle 1-122
3EH Close File Handle 1-124
3FH Read File or Device 1-126
40H Write File or Device 1-128
41H Delete Entry 1-130
42H Move Pointer 1-132
43H Change Mode 1-134
44H I/O Control for Devices 1-136
Codes 00H and 01H 1-137
Codes 02H, 03H, 04H, and 05H 1-138
Codes 06H and 07H 1-138
45H Duplicate a File Handle 1-140
46H Force Duplicate File Handle 1-142
47H Get Current Directory 1-144
48H Allocate Memory 1-146
49H Free Allocated Memory 1-148
4AH Set Block 1-150
4BH Execute Program 1-152
Code 00H Load and Execute Program 1-152
Code 03H Load But Don't Execute 1-154

4CH End Process 1-156
4DH Wait 1-158
4EH Find First File 1-159
4FH Find Next File 1-162
54H Get/Verify State 1-164
56H Rename a File 1-165
57H Get/Set Date or Time 1-167
1.9 Macro Definitions for MS-DOS System 1-169

CHAPTER 1

SYSTEM CALLS

1.1 INTRODUCTION

MS-DOS lets your programs call system routines to manage files, devices, system resources, and other programs. These system calls save you the trouble of writing your own system-management routines, make it easier to write machine-independent programs, and increase the likelihood that your programs will be compatible with future versions of MS-DOS.

There are four general types of system calls:

1. Input and output to standard character devices: Read a character, write a character, read a string, write a string, print a character.

2. File management: Create, open, read, write, close, rename, and delete a file; create or delete a directory, change the current directory.

3. Program management: End a program, keep a program in memory, load and execute another program, allocate more memory, release memory.

4. Miscellaneous system management: Get or set the time or date, get the MS-DOS version number, etc.

System calls include interrupts, which are serviced by interrupt handlers, and function requests, which call MS-DOS routines. Version 2.0 introduces several new system calls, some of which replace earlier ones; the old ones are still supported to maintain compatibility with earlier versions of MS-DOS, but you should use the new calls unless

circumstances demand that you maintain backward compatibility.

The introduction to this chapter describes the difference between the old and new system calls, how to execute a system call, how to return control to MS-DOS, the format of a directory entry, and how each call is described later in the chapter. For those who must use the old calls, a description of the File Control Block appears toward the end of the introduction. The introduction concludes with tables that list the interrupts and function requests in numeric and alphabetic order.

The remainder of the chapter consists of a description of each interrupt and function request (including a sample program), a listing of the macro definitions used in the sample programs, and an extended sample program.

1.1.1 About Interrupts

MS-DOS reserves interrupts 20H through 3FH for its own use. The table of interrupt routine addresses (vectors) is maintained in locations 80H-FCH. Most of the interrupts have been superseded by function requests. Descriptions of three MS-DOS interrupt handlers (program terminate, CONTROL-C, and fatal error) are included in case you must write your own routines to handle these interrupts.

1.1.2 About Function Requests

The function requests call MS-DOS routines to manage system resources. As described in the following section, many calls added in Version 2.0 supersede older calls, particularly in the areas of file handling and program management; some of these calls are compatible with the XENIX(TM) operating system. Wherever two calls provide essentially the same service, you should use the newer calls unless your programs must be backward-compatible with earlier versions of MS-DOS.

1.2 CALLS COMPATIBLE WITH THE XENIX OPERATING SYSTEM

Version 2.0 adds function requests that provide similar capabilities to earlier calls but are compatible with the XENIX operating system; they implement the hierarchical filing system and make it simpler to manage system resources and return control to MS-DOS. Table 1.1 lists these calls.

Table 1.1 Function Requests Compatible With the XENIX Operating System

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

Unless it's imperative that your program be compatible with pre-2.0 versions of MS-DOS, you should use the calls listed in Table 1.1. If you do use the older calls, your program will run under version 2.0, but it will not be able to use the features added by version 2.0.

1.3 USING THE SYSTEM CALLS

1.3.1 Issuing An Interrupt

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

1.3.2 Calling A Function Request

If your program has a standard Program Segment Prefix, follow this procedure to call a function request:

1. Move any required data into the registers.
2. Move the function number into AH.

3. Execute a long call to location 50H in the Program Segment Prefix.

If your program doesn't have a standard Program Segment Prefix, follow steps 1 and 2, then issue Interrupt 21H.

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

1.3.3 Using The Calls From A High-Level Language

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

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

Use the CALL statement or USER function to execute the required 8086 code from the BASIC interpreter.

1.3.4 Treatment Of Registers

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

1.3.5 Error Checking

Many system calls return a value in a register that specifies whether the operation was successful. Your program can handle error conditions by checking the register that contains the error code (usually AX) and taking appropriate action.

The calls compatible with the XENIX operating system also

set the Carry flag if there is an an error, and identify the specific error by returning a number in AX. Table 1.2 lists these error codes and their meaning.

Table 1.2 Error Return Codes

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

To handle error conditions, put the following statement immediately following each call compatible with the XENIX operating system:

```
JC <error>
```

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

1.3.6 Returning Control To MS-DOS

To return control to MS-DOS, call Function Request 4CH. If you want the program to remain in memory, call Function Request 31H instead.

If you must maintain backward compatibility with earlier versions of MS-DOS, you can return control to MS-DOS by issuing Interrupt 20H, calling Function Request 00H, or jumping to location 00H in the Program Segment Prefix. To keep the program in memory, issue Interrupt 27H. If you use any of these techniques, CS must contain the segment address of the Program Segment Prefix.

1.4 HIERARCHICAL DIRECTORY SYSTEM

MS-DOS version 2.0 supports a hierarchical (multi level) directory system, similar to the XENIX operating system. (For information on using the multi level directory system, see the MS-DOS 2.0 User's Guide.)

Version 2.0 treats the file as a string of bytes, with no notion of records, record length, or access technique (sequential or random). Your programs impose a record structure on this string of bytes, giving you greater flexibility in handling files and simplicity in dealing with MS-DOS.

Table 1.3 lists file-related functions, the function requests compatible with the XENIX operating system and the corresponding old function request numbers.

Table 1.3 New and Old File-Related Function Requests

Function Request	Compatible With XENIX	Old
Create a file	3CH	16H
Open a file	3DH	0FH
Close a file	3EH	10H
Read from a file	3FH	14H, 21H, 27H
Write to a file	40H	15H, 22H, 28H
Delete a file	41H	13H
Move file pointer	42H	24H
Search directory for first matching entry	4EH	11H
Search directory for next matching entry	4FH	12H
Rename a file	56H	17H

MS-DOS does not restrict the depth of a directory structure except in the number of allocation units available. The root directory has a fixed number of entries (64 for a single-sided disk, 112 for a double-sided disk). The number of files in all other directories is limited only limited by the number of allocation units available.

Pre-2.0 disks appear to MS-DOS 2.0 as having only a root directory that contains files and no subdirectories.

Implementation of the multi level directory structure is simple. The root directory is the pre-2.0 directory. Each subdirectory has a special attribute set indicating that it is a directory. The subdirectories themselves are files, linked through the FAT as usual. Their entries are identical to the entries in the root directory.

Pre-2.0 directories that use system calls not described in

this chapter will be unable to make use of files in other directories. Those files not necessary for the current task will be placed in other directories.

1.4.1 Handles

To create or open a file, you specify a path that identifies the drive, directory, name, and extension of the file. MS-DOS returns a number called a handle. In most cases, subsequent actions on this file refer to it only by its handle; there is no FCB to maintain.

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

Table 1.4 Predefined Device Handles

Handle	Standard device
00H	Input
01H	Output
02H	Error
03H	Auxiliary
04H	Printer

Handles 00H and 01H (standard input and output) can be redirected.

1.4.2 Directory Entry

A directory contains one entry for each file on the disk. Each entry is 32 bytes. Table 1.5 describes the fields of an entry.

Table 1.5 Format of the Directory Entry

Name	Size (bytes)	Offset Hex	Offset Decimal
Filename	8	00H	0
Extension	3	08H	8
Attributes	1	0BH	11
RESERVED	10	0CH	12
Time of last write	2	16H	22
Date of last write	2	18H	24
RESERVED	2	1AH	26
File size	4	1CH	28

1.4.3 Fields Of The Directory Entry

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

00H	End of allocated directory
E5H	Free directory entry

Extension (offset 08H): Three characters, left-aligned and padded (if necessary) with blanks. If there is no extension, this field is all blanks.

Attributes (offset 0BH): File attributes (see Table 1.6).

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

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

```
Offset 17H          Offset 16H

|H|H|H|H|H|M|M|M|   |M|M|M|S|S|S|S|S|
15      11 10           5 4       0
```

Date of Last Write (offset 18H): The date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

```
Offset 19H          Offset 18H

|Y|Y|Y|Y|Y|Y|Y|M|   |M|M|M|D|D|D|D|D|
15            9 8         5 4       0
```

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

1.4.4 File Attributes

Table 1.6 describes the file attributes and how they are represented in the attribute byte of the directory entry (offset 0BH).

Table 1.6 File Attributes

Attribute	Code	Comment
Normal	00H	Can be read or written without restriction.
Read-only	01H	Attempts to open the file for write or create a file with the same name fails.
Hidden	02H	File not found by directory search. Old Open File (Function Request 0FH) fails.
System	04H	File not found by directory search. Old Open File (Function Request 0FH) fails.
Volume-ID	08H	Only one file (in the root directory) can have this attribute.
Directory	10H	Identifies a subdirectory. Cannot be changed with Function Request 43H (Change Mode).
Archive	20H	Set when the file is written.

1.5 SYSTEM CALL DESCRIPTIONS

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

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

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

A description of the processing performed.

A more complete description of the register contents after the system call.

An example of its use.

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

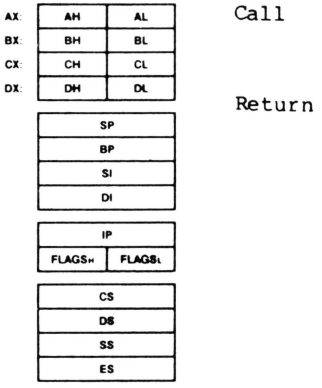


Figure 1.1 Example of System Call Description

1.5.1 Sample Programs

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

```
code        segment
            assume  cs:code,ds:code,es:nothing,ss:nothing
            org     100H
start:      mov     word ptr loc_50h[2],cs
            jmp     begin
;
loc_50h     label   dword
            dw      50h
            dw      ?
;
filename    db      "b:\textfile.asc",0
buffer      db      129 dup (?)
handle      dw      ?
;
begin:      open_handle filename,0     ; Open the file
            jc      error_open         ; Routine not shown
            mov     handle,ax          ; Save handle
read_line:  read_handle handle,buffer,128 ; Read 128 bytes
            jc      error_read         ; Routine not shown
            cmp     ax,0               ; End of file?
            je      return             ;   Yes, go home
            mov     bx,ax              ;   No, AX bytes read
            mov     buffer[bx],"$"     ; To terminate string
            display buffer             ; See Function 09H
            jmp     read_line          ; Get next 128 bytes

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

Figure 1.2 Sample Program With Common Skeleton

To allow the examples to be more complete programs rather than isolated uses of the system calls, a macro is defined for each system call. These macros are used in the sample programs; OPEN-HANDLE, READHANDLE, DISPLAY, and ENDPROCESS in Figure 1.2 are such macros. A few general-purpose macros are also used. All macro definitions are listed at the end of this chapter.

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

1.5.2 Error Handling

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

1.6 FILE CONTROL BLOCK (FCB)

The old function requests that deal with files require that your program build and maintain a File Control Block (FCB) for each file that contains its name, size, record length, pointer to current record, etc. Each old function request that refers to a file must have access to an up-to-date version of the file's FCB.

The descriptions refer to unopened and opened FCBs. An unopened FCB contains only a drive specifier and filename. An opened FCB contains all fields filled by the Open File system call (Function 0FH).

The Program Segment Prefix includes room for two FCBs at offsets 5CH and 6CH. See Chapter 4 for a description of the PSP and how these FCBs are used. Table 1.7 describes the fields of the FCB.

Table 1.7 Format of the File Control Block (FCB)

Name	Size (bytes)	Offset Hex	Offset Decimal
Drive number	1	00H	0
Filename	8	01H	1
Extension	3	08H	9
Current block	2	0BH	12
Record size	2	0DH	14
File size	4	0FH	16
Date of last write	2	13H	20
Time of last write	2	15H	22
RESERVED	8	17H	24
Current record	1	1FH	32
Relative record	4	20H	33

1.6.1 Fields Of The FCB

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

Filename (offset 01H): Eight characters, left-aligned and padded (if necessary) with blanks. If you specify a reserved device name (such as LPT1), do not put a colon at the end.

Extension (offset 09H): Three characters, left-aligned and padded (if necessary) with blanks. This field can be all

blanks (no extension).

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

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

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

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

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

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

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

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

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

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

NOTE

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

1.6.2 Extended FCB

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

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

Tables 1.8 through 1.11 list the Interrupts and Function Requests in numeric and alphabetic order.

Table 1.8 MS-DOS Interrupts, Numeric Order

Interrupt	Description
20H	Program Terminate
21H	Function Request
22H	Terminate Address
23H	CONTROL-C Exit Address
24H	Fatal Error Abort Address
25H	Absolute Disk Read
26H	Absolute Disk Write
27H	Terminate But Stay Resident
28H-40H	RESERVED

Table 1.9 MS-DOS Interrupts, Alphabetic Order

Description	Interrupt
Absolute Disk Read	25H
Absolute Disk Write	26H
CONTROL-C Exit Address	23H
Fatal Error Abort Address	24H
Function Request	21H
Program Terminate	20H
RESERVED	
Terminate Address	22H
Terminate But Stay Resident	27H

Table 1.10 MS-DOS Function Requests, Numeric Order

Function	Description
00H	Terminate Program
01H	Read Keyboard And Echo
02H	Display Character
03H	Auxiliary Input
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
07H	Direct Console Input
08H	Read Keyboard
09H	Display String
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
0DH	Reset Disk
0H	Select Disk
0FH	Open File
10H	Close File
11H	Search For First Entry
12H	Search For Next Entry
13H	Delete File
14H	Sequential Read
15H	Sequential Write
16H	Create File
17H	Rename File
18H	RESERVED
19H	Current Disk
1AH	Set Disk Transfer Address
1BH-20H	RESERVED
21H	Random Read
22H	Random Write
23H	File Size
24H	Set Relative Record
25H	Set Vector
26H	Create New Program Segment
27H	Random Block Read
28H	Random Block Write
29H	Parse File Name
2AH	Get Date
2BH	Set Date
2CH	Get time
2DH	Set Time
2EH	Set/Reset Verify Flag
2FH	Get Disk Transfer Address
30H	Get MS-DOS Version Number
31H	Keep Process
32H	RESERVED
33H	CONTROL-C Check
34H	RESERVED
35H	Get Interrupt Vector
36H	Get Disk Free Space

37H	RESERVED
38H	Return Country-Dependent Information
39H	Create Sub-Directory
3AH	Remove A Directory Entry
3BH	Change Current Directory
3CH	Create A File
3DH	Open A File
3H	Close A File Handle
3FH	Read From File Or Device
40H	Write To A File Or Device
41H	Delete A Directory Entry
42H	Move A File Pointer
43H	Change Attributes
44H	I/O Control For Devices
45H	Duplicate A File Handle
46H	Force A Duplicate Of A Handle
47H	Return Text Of Current Directory
48H	Allocate Memory
49H	Free Allocated Memory
4AH	Modify Allocated Memory Blocks
4BH	Load And Execute A Program
4CH	Terminate A Process
4DH	Return The Return Code Of A Child
4H	Find Match File
4FH	Step Through A Directory Matching Files
50H-53H	RESERVED
54H	Return Current Setting Of Verify
55H	RESERVED
56H	Move a Directory Entry
57H	Get/Set Date/Time Of File

Table 1.11 MS-DOS Function Requests, Alphabetic Order

Function	Description
48H	Allocate Memory
03H	Auxiliary Input
04H	Auxiliary Output
0AH	Buffered Keyboard Input
43H	Change Attributes
3BH	Change Current Directory
0BH	Check Keyboard Status
3H	Close A File Handle
10H	Close File
33H	CONTROL-C Check
3CH	Create A File
16H	Create File
26H	Create New Program Segment
39H	Create Sub-Directory
19H	Current Disk
41H	Delete A Directory Entry
13H	Delete File
07H	Direct Console Input
06H	Direct Console I/O
02H	Display Character
09H	Display String
45H	Duplicate A File Handle
23H	File Size
4EH	Find Match File
0CH	Flush Buffer, Read Keyboard
46H	Force A Duplicate of a Handle
49H	Free Allocated Memory
2AH	Get Date
36H	Get Disk Free Space
2FH	Get Disk Transfer Address
30H	Get MS-DOS Version Number
35H	Get Interrupt Vector
2CH	Get Time
57H	Get/Set Date/Time Of File
44H	I/O Control For Devices
31H	Keep Process
4BH	Load And Execute A Program
4AH	Modify Allocated Memory Blocks
56H	Move A Directory Entry
42H	Move A File Pointer
18H	RESERVED
1BH-20H	RESERVED
32H	RESERVED
34H	RESERVED
37H	RESERVED
50H-53H	RESERVED
55H	RESERVED
3DH	Open A File
0FH	Open File
29H	Parse File Name

05H	Print Character
27H	Random Block Read
28H	Random Block Write
21H	Random Read
22H	Random Write
3FH	Read From File Or Device
08H	Read Keyboard
01H	Read Keyboard And Echo
3AH	Remove A Directory Entry
17H	Rename File
0DH	Reset Disk
38H	Return Country-Dependent Information
54H	Return Current Setting Of Verify
47H	Return Text Of Current Directory
4DH	Return The Return Code Of A Child
11H	Search For First Entry
12H	Search For Next Entry
0EH	Select Disk
14H	Sequential Read
15H	Sequential Write
2BH	Set Date
1AH	Set Disk Transfer Address
24H	Set Relative Record
2DH	Set Time
25H	Set Vector
2EH	Set/Reset Verify Flag
4FH	Step Through A Directory Matching Files
4CH	Terminate A Process
00H	Terminate Program
40H	Write To A File Or Device

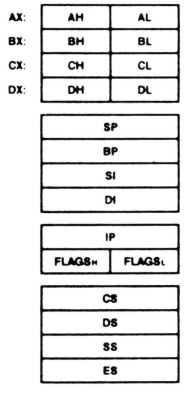
A detailed description of each system call follows. They are listed in numeric order; the interrupts are described first, then the function requests.

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

1.7 INTERRUPTS

The following pages describe Interrupts 20H--27H.

Program Terminate (Interrupt 20H)



Call
CS
Segment address of Program Segment Prefix

Return
None

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

The CS register must contain the segment address of the Program Segment Prefix before you call this interrupt. The following exit addresses are restored from the Program Segment Prefix:

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

All file buffers are flushed to disk.

NOTE

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

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

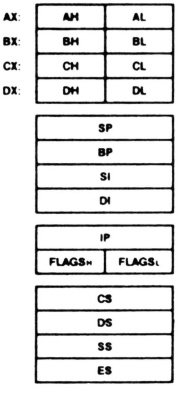
Macro Definition:
```
terminate macro
          int 20H
          endm
```

Example

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

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

Function Request (Interrupt 21H)



Call
AH
  Function number
Other registers as specified in
  individual function

Return
As specified in individual function

The AH register must contain the number of the system function. See Section 1.8, "Function Requests," for a description of the MS-DOS system functions.

NOTE

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

Example

To call the Get Time function:

```
mov   ah,2CH          ;Get Time is Function 2CH
call  loc_50H         ;THIS INTERRUPT
```

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

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

Interrupt 22H -- Terminate Address

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

Interrupt 23H -- CONTROL-C Exit Address

If the user types CONTROL-C during keyboard input or display output, control transfers to the INT 23H vector in the interrupt table. This address is copied into the Program Segment Prefix, from the Interrupt 23H vector, when the segment is created. If the CONTROL-C routine preserves all registers, it can end with an IRET instruction (return from interrupt) to continue program execution. When the interrupt occurs, all registers are set to the value they had when the original call to MS-DOS was made. There are no restrictions on what a CONTROL-C handler can do -- including MS-DOS function calls -- so long as the registers are unchanged if IRET is used. If Function 09H or 0AH (Display String or Buffered Keyboard Input) is interrupted by CONTROL-C, the three-byte sequence 03H-0DH-0AH (ETX-CR-LF) is sent to the display and the function resumes at the beginning of the next line. If the program creates a new segment and loads a second program that changes the CONTROL-C address, termination of the second program restores the CONTROL-C address to its value before execution of the second program.

Interrupt 24H -- Fatal Error Abort Address

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

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

NOTE

Interrupt 24H is not issued if the failure occurs during execution of Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write). These errors are usually handled by the MS-DOS error routine in COMMAND.COM that retries the disk operation, then gives the user the choice of aborting, retrying the operation, or ignoring the error. The following topics give you the information you need about interpreting the error codes, managing the registers and stack, and controlling the system's response to the error in order to write your own error-handling routines.

Error Codes

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

The following are error codes for Interrupt 24H:

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

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

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

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

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

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

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

Disk Errors

If the AH bit 7 = 0 (there is a hard error on a disk), register AL contains the failing drive number, with 0 = drive A. AH bits 0-2 indicate the disk area and whether the fail occurred during a read or write operation, as follows:

Bit0=0 if read operation
1 if write operation

Bits 2-1 (affected disk area)

0 0 MS-DOS area (system files)
0 1 File Allocation Table
1 0 directory
1 1 data area

Other Errors

If AH bit 7 = 1, then the error occurred on a character device or resulted from a bad memory image of the File Allocation Table. You can examine the device header passed in BP:SI to determine the exact cause. 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 the error was on a character device, the contents of AL are unpredictable, and the error code is returned in DI.

Notes:

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

2. This routine is entered in a disabled state.

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

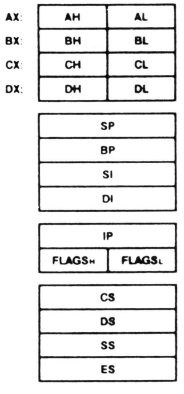
4. This interrupt handler should refrain from using MS-DOS function calls. If necessary, it may use calls 01H through 0CH. Use of any other call will destroy the MS-DOS stack and will leave MS-DOS in an unpredictable state.

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

6. If the interrupt handler will handle errors rather than returning to MS-DOS, it should restore the application program's registers from the stack, remove all but the last three words on the stack,

then issue an IRET. This will return to the program immediately after the INT 21H that experienced the error. Note that if this is done, MS-DOS will be in an unstable state until a function call higher than 0CH is issued.

Absolute Disk Read (Interrupt 25H)



Call

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

Return

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

The registers must contain the following:

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

This interrupt transfers control to the MS-DOS BIOS. The number of sectors specified in CX is read from the disk to the Disk Transfer Address. Its requirements and processing are identical to Interrupt 26H, except data is read rather than written.

NOTE

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

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

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

Macro Definition:

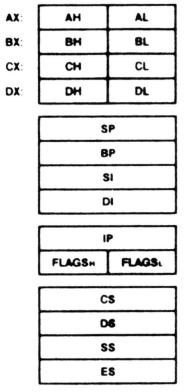
```
abs_disk_read  macro  disk,buffer,num_sectors,first_sector
               mov    al,disk
               mov    bx,offset buffer
               mov    cx,num_sectors
               mov    dx,first_sector
               int    25H
               popf
               endm
```

Example

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

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

Absolute Disk Write (Interrupt 26H)



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

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

The registers must contain the following:

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

This interrupt transfers control to MS-DOS. The number of sectors specified in CX is written from the Disk Transfer Address to the disk. Its requirements and processing are identical to Interrupt 25H, except data is written to the disk rather than read from it.

NOTE

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

The system pushes the flags at the time of the call; they are still there upon return. Be sure to pop the stack upon return to prevent uncontrolled growth. If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H for the codes and their meaning).

Macro Definition:

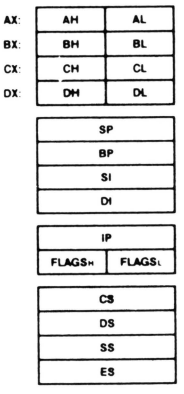
```
abs_disk_write  macro  disk,buffer,num_sectors,first_sector
                mov    al,disk
                mov    bx,offset buffer
                mov    cx,num_sectors
                mov    dx,first_sector
                int    26H
                popf
                endm
```

Example

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

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

Terminate But Stay Resident (Interrupt 27H)



Call
CS:DX
First byte following
last byte of code

Return
None

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

DX must contain the offset (from the segment address in CS) of the first byte following the last byte of code in the program. When Interrupt 27H is executed, the program terminates but is treated as an extension of MS-DOS; it remains resident and is not overlaid by other programs when it terminates.

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

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

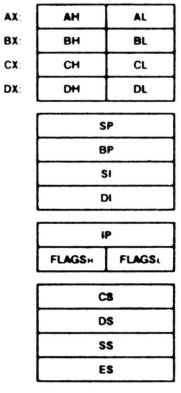
Example

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

1.8 FUNCTION REQUESTS

The following pages describe function calls 00H-57H.

Terminate Program (Function 00H)



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

Return
None

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

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

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

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

All file buffers are flushed to disk.

Warning

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

Macro Definition:

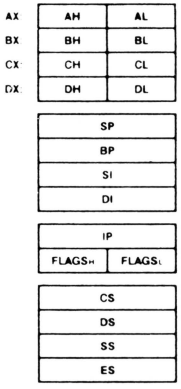
```
terminate_program  macro
                   xor     ah,ah
                   call    loc_50H
                   endm
```

Example

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

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

Read Keyboard and Echo (Function 01H)



Call
AH = 01H

Return
AL
Character typed

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

Macro Definition:

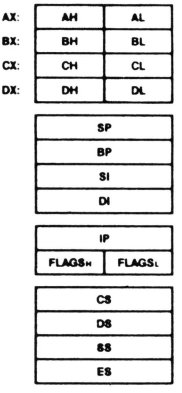
```
read_kbd_and_echo  macro
                   mov  ah, 01H
                   call  loc_50H
                   endm
```

Example

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

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

Display Character (Function 02H)



Call
AH = 02H
DL
Character to be displayed

Return
None

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

Macro Definition:

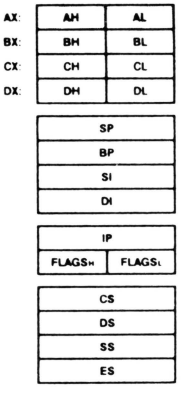
```
display_char macro   character
             mov     dl,character
             mov     ah,02H
             int     21H
             endm
```

Example

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

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

Auxiliary Input (Function 03H)



Call
AH = 03H

Return
AL
Character from auxiliary device

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

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

Macro Definition:

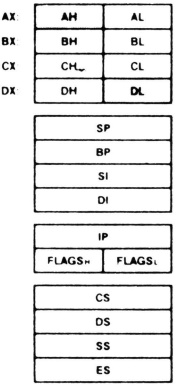
```
aux_input  macro
           mov  ah,03H
           call loc_50H
           endm
```

Example

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

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

Auxiliary Output (Function 04H)



Call
AH = 04H
DL
Character for auxiliary device

Return
None

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

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

Macro Definition:

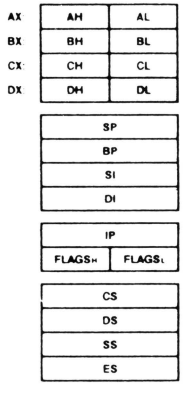
```
aux_output  macro  character
            mov  dl,character
            mov  ah,04H
            call  loc_50H
            endm
```

Example

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

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

Print Character (Function 05H)



Call
AH = 05H
DL
Character for printer

Return
None

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

Macro Definition:

```
print_char macro character
           mov   dl,character
           mov   ah,05H
           call  loc_50H
           endm
```

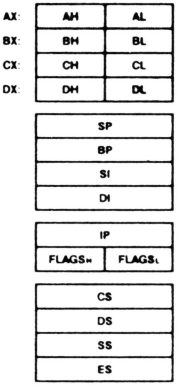
Example

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

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

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

Direct Console I/O (Function 06H)



Call
AH = 06H
DL
See text

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

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

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

DL is not FFH -- The character in DL is displayed.

This function does not check for CONTROL-C.

Macro Definition:

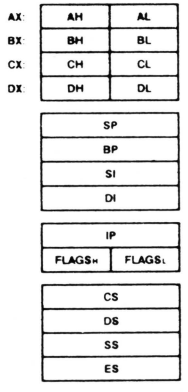
```
dir_console_io  macro switch
                mov  dl,switch
                mov  ah,06H
                call loc_50H
                endm
```

Example

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

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

Direct Console Input (Function 07H)



Call
AH = 07H

Return
AL
Character from keyboard

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

Macro Definition:

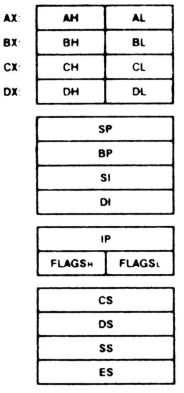
```
dir_console_input  macro
                   mov  ah,07H
                   call  loc_50H
                   endm
```

Example

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

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

Read Keyboard (Function 08H)



Call
AH = 08H

Return
AL
Character from keyboard

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

Macro Definition:

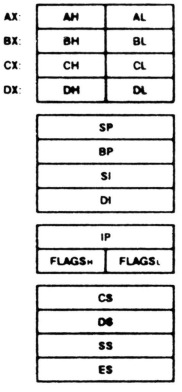
```
read_kbd  macro
          mov   ah,08H
          call  loc_50H
          endm
```

Example

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

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

Display String (Function 09H)



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

Return
None

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

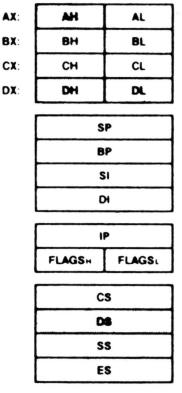
Macro Definition:
```
display  macro string
         mov   dx,offset string
         mov   ah,09H
         call  loc_50H
         endm
```

Example

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

```
table    db      "0123456789ABCDEF"
sixteen  db       10H
result   db      " - 00H",0DH,0AH,"$"   ;see text for
                                        ;explanation of $
begin:   read_kbd_and_echo              ;see Function 01H
         convert al,sixteen,result[3]   ;see end of chapter
         display result                 ;THIS FUNCTION
         jmp     begin                  ;do it again
```

Buffered Keyboard Input (Function 0AH)



Call
AH = 0AH
DS:DX
  Input buffer

Return
None

DX must contain the offset (from the segment address in DS) of an input buffer of the following form:

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

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

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

Macro Definition:

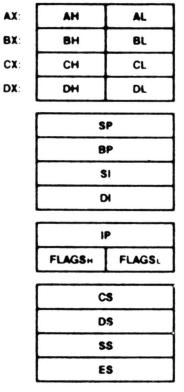
```
get_string  macro  limit,string
            mov    dx,offset string
            mov    string,limit
            mov    ah,0AH
            call   loc_50H
            endm
```

Example

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

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

Check Keyboard Status (Function 0BH)



Call
AH = 0BH

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

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

Macro Definition:

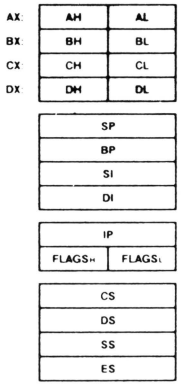
```
check_kbd_status  macro
                  mov   ah,0BH
                  call loc_50H
                  endm
```

Example

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

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

Flush Buffer, Read Keyboard (Function 0CH)



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

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

The keyboard type-ahead buffer is emptied. Further processing depends on the value in AL when the function is called.

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

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

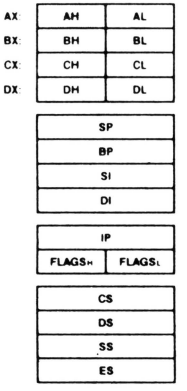
Macro Definition:
```
flush_and_read_kbd  macro switch
                    mov   al,switch
                    mov   ah,0CH
                    call  loc_50
                    endm
```

Example

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

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

Disk Reset (Function 0DH)



Call
AH = 0DH

Return
None

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

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

Macro Definition:

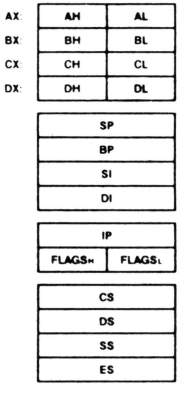
```
reset_disk macro
           mov   ah,0DH
           call  loc_50H
           endm
```

Example

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

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

Select Disk (Function 0EH)



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

Return
AL
  Number of logical drives

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

NOTE

For future compatibility the value returned in AL must be treated with care. For instance, if AL returns 5 it is not safe to assume drives A, B, C, D, E are all valid drive designators.

Macro Definition:

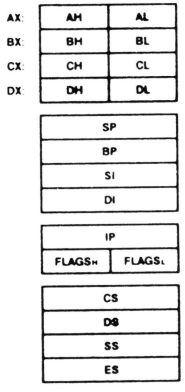
```
select_disk macro disk
            mov   dl,disk[-64]
            mov   ah,0EH
            call  loc_50H
            endm
```

Example

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

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

Open File (Function 0FH)



Call
AH = 0FH
DS:DX
  Unopened FCB

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

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

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

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

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

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

Macro Definition:

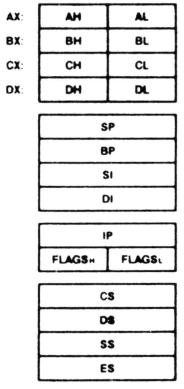
```
open macro fcb
     mov   dx,offset fcb
     mov   ah,0FH
     call  loc_50H
     endm
```

Example

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

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

Close File (Function 10H)



Call
AH = 10H
DS:DX
  Opened FCB

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

DX must contain the offset (to the segment address in DS) of an opened FCB. The disk directory is searched for the file named in the FCB. This function must be called after a file is changed to update the directory entry. It is strongly advised that any FCB (even one for a file that hasn't been changed) be closed when access to the file is no longer needed.

If a directory entry for the file is found, the location of the file is compared with the corresponding entries in the FCB. The directory entry is updated, if necessary, to match the FCB, and AL returns 0.

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

Macro Definition:
```
close macro fcb
      mov   dx,offset fcb
      mov   ah,10H
      call  loc_50H
      endm
```

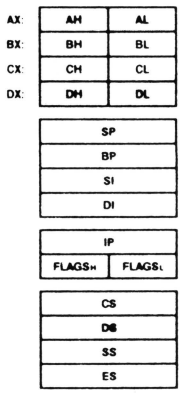
Example

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

```
message         db    "Not saved in ASCII format",0DH,0AH,"$"
fcb             db     2,"MOD1    BAS"
                db     26  dup (?)
buffer          db     128 dup (?)
;
begin:          set_dta  buffer        ;see Function 1AH
```

```
                open  fcb               ;see Function 0FH
                read_seq  fcb           ;see Function 14H
                cmp   buffer,0FFH        ;is first byte FFH?
                jne   all_done          ;no
                display  message        ;see Function 09H
all_done:       close fcb               ;THIS FUNCTION
```

Search for First Entry (Function 11H)



Call
AH = 11H
DS:DX
  Unopened FCB

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

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

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

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

The original FCB at DS:DX contains information to continue the search with Function 12H. This FCB must not be altered or opened.

Notes:

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

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

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

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

Macro Definition:

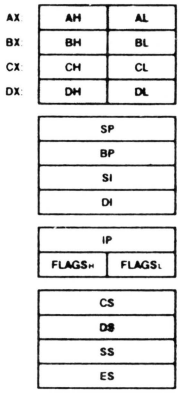
```
search_first macro fcb
             mov   dx,offset fcb
             mov   ah,11H
             call  loc_50H
             endm
```

Example

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

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

Search for Next Entry (Function 12H)



Call
AH = 12H
DS:DX
  Unopened FCB

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

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

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

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

Macro Definition:
```
search_next macro fcb
            mov   dx,offset fcb
            mov   ah,12H
            call  loc_50H
            endm
```

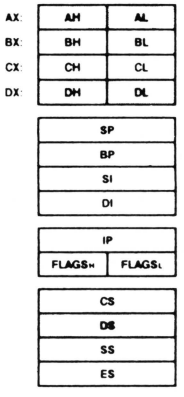
Example

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

```
message     db    "No files",0DH,0AH,"$"
files       db     0
ten         db     0AH
fcb         db     2,"???????????"
            db     26  dup (?)
buffer      db     128 dup (?)
```

```
;
begin:      set_dta  buffer          ;see Function 1AH
            search_first  fcb        ;see Function 11H
            cmp  al,0FFH             ;directory entry found?
            je   all_done            ;no, no files on disk
            inc  files               ;yes, increment file
                                     ;counter
search_dir: search_next  fcb         ;THIS FUNCTION
            cmp  al,0FFH             ;directory entry found?
            je   done                ;no
            inc  files               ;yes, increment file
                                     ;counter
            jmp  search_dir          ;check again
done:       convert files,ten,message ;see end of chapter
all_done:   display  message         ;see Function 09H
```

Delete File (Function 13H)



Call
AH = 13H
DS:DX
  Unopened FCB

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

DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for a matching filename. The filename in the FCB can contain the (?) wild card character to match any character.

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

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

Macro Definition:
```
delete  macro  fcb
        mov    dx,offset fcb
        mov    ah,13H
        call   loc_50H
        endm
```

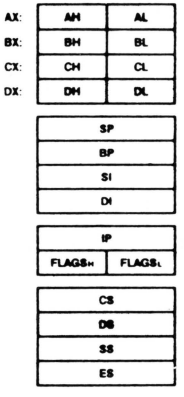
Example

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

```
year            dw    1982
month           db    12
day             db    31
files           db    0
ten             db    0AH
message         db    "NO FILES DELETED.",0DH,0AH,"$"
fcb             db    2,"???????????"
                db    26 dup  (?)
buffer          db    128 dup (?)
;
```

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

Sequential Read (Function 14H)



Call
AH = 14H
DS:DX
  Opened FCB

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

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

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

AL returns a code that describes the processing:

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

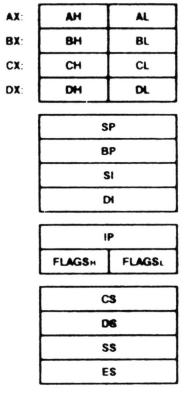
Macro Definition:
```
read_seq macro fcb
         mov   dx,offset fcb
         mov   ah,14H
         call  loc_50H
         endm
```

Example

The following program displays the file named TEXTFILE.ASC that is on the disk in drive B; its function is similar to the MS-DOS TYPE command. If a partial record is in the buffer at end of file, the routine that displays the partial record displays characters until it encounters an end-of-file mark (ASCII 26, or CONTROL-Z).

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

Sequential Write (Function 15H)



Call
AH = 15H
DS:DX
  Opened FCB

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

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

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

AL returns a code that describes the processing:

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

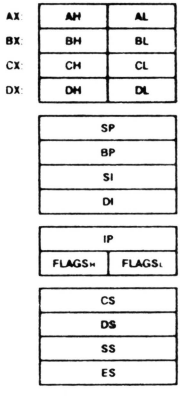
Macro Definition:
```
write_seq macro fcb
          mov   dx,offset fcb
          mov   ah,15H
          call  loc_50H
          endm
```

Example

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

```
record_size     equ   0EH            ;offset of Record Size
;                                    field in FCB
fcb1            db    2,"DIR     TMP"
                db    26 dup (?)
fcb2            db    2,"???????????"
                db    26 dup (?)
buffer          db    128 dup (?)
;
begin:      set_dta       buffer      ;see Function 1AH
            search_first  fcb2        ;see Function 11H
            cmp           al,0FFH     ;directory entry found?
            je            all_done    ;no, no files on disk
            create        fcb1        ;see Function 16H
            mov           fcb1[record_size],12
                                      ;set record size to 12
write_it:   write_seq     fcb1        ;THIS FUNCTION
            cmp           al,0        ;write successful?
            jne           all_done    ;no, go home
            search_next   fcb2        ;see Function 12H
            cmp           al,FFH      ;directory entry found?
            je            all_done    ;no, go home
            jmp           write_it    ;yes, write the record
all_done:   close         fcb1        ;see Function 10H
```

Create File (Function 16H)



Call
AH = 16H
DS:DX
  Unopened FCB

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

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

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

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

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

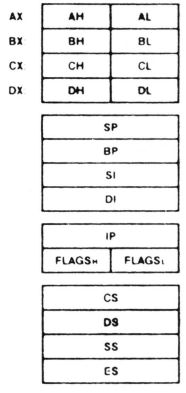
Macro Definition:
```
create macro fcb
       mov   dx,offset fcb
       mov   ah,16H
       call  loc_50H
       endm
```

Example

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

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

Rename File (Function 17H)



Call
AH = 17H
DS:DX
Modified FCB

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

DX must contain the offset (from the segment address in DS) of an FCB with the drive number and filename filled in, followed by a second filename at offset 11H. The disk directory is searched for an entry that matches the first filename, which can contain the (?) wild card character.

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

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

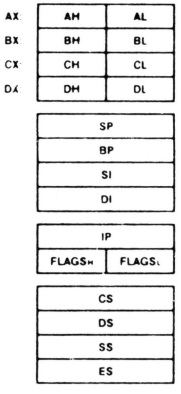
Macro Definition:
```
rename macro fcb,newname
        mov   dx,offset fcb
        mov   ah,17H
        call  loc_50H
        endm
```

Example

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

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

Current Disk (Function 19H)



Call
AH = 19H

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

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

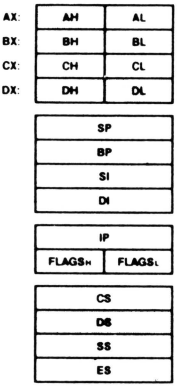
Macro Definition:
```
current_disk  macro
              mov   ah,19H
              call  loc_50H
              endm
```

Example

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

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

Set Disk Transfer Address (Function lAH)



Call
AH = lAH
DS:DX
Disk Transfer Address

Return
None

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

NOTE

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

Macro Definition:

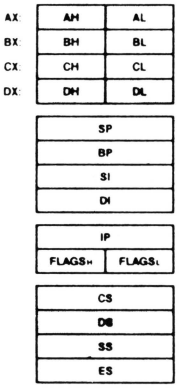
```
set_dta  macro  buffer
         mov    dx,offset buffer
         mov    ah,lAH
         call   loc_50H
         endm
```

Example

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

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

Random Read (Function 21H)



Call
AH = 21H
DS:DX
  Opened FCB

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

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

AL returns a code that describes the processing:

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

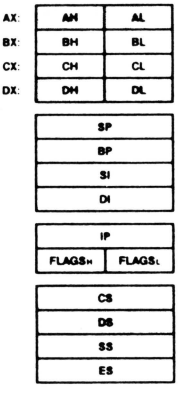
Macro Definition:
```
read_ran  macro  fcb
          mov    dx,offset fcb
          mov    ah,21H
          call   loc_50H
          endm
```

Example

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

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

Random Write (Function 22H)



Call
AH = 22H
DS:DX
  Opened FCB

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

DX must contain the offset from the segment address in DS of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is written from the Disk Transfer Address. If the Record Size is less than a sector, the data at the Disk Transfer Address is written to a buffer; the buffer is written to disk when it contains a full sector of data, or the file is closed, or a Reset Disk system call (Function 0DH) is issued.

AL returns a code that describes the processing:

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

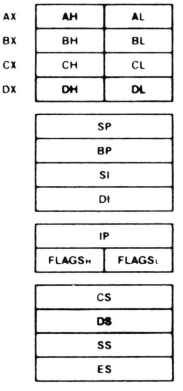
Macro Definition:
```
write_ran  macro  fcb
           mov    dx,offset fcb
           mov    ah,22H
           call   loc_50H
           endm
```

Example

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

```
record_size     equ   0EH       ;offset of Record Size
                                 ;field of FCB
relative_record equ   21H       ;offset of Relative Record
;                                ;field of FCB
fcb         db    2,"ALPHABETDAT"
            db    26  dup (?)
buffer      db    28 dup(?),0DH,0AH,"$"
promptl     db   "Enter letter: $"
prompt2     db   "New record (RETURN for no change): $"
crlf        db    0DH,0AH,"$"
reply       db    28 dup (32)
blanks      db    26 dup (32)
;
begin:      set_dta  buffer              ;see Function 1AH
            open     fcb                 ;see Function 0FH
            mov      fcb[record_size],28 ;set record size
get_char:   display  promptl             ;see Function 09H
            read_kbd_and_echo            ;see Function 01H
            cmp      al,0DH              ;just a CR?
            je       all_done            ;yes, go home
            sub      al,41H              ;convert ASCII
                                         ;code to record #
            mov      fcb[relative_record],al
                                         ;set relative record
            display  crlf                ;see Function 09H
            read_ran fcb                 ;THIS FUNCTION
            display  buffer              ;see Function 09H
            display  crlf                ;see Function 09H
            display  prompt2             ;see Function 09H
            get_string 27,reply          ;see Function 0AH
            display  crlf                ;see Function 09H
            cmp      reply[1],0          ;was anything typed
                                         ;besides CR?
            je       get_char            ;no
                                         ;get another char.
            xor      bx,bx               ;to load a byte
            mov      bl,reply[1]         ;use reply length as
                                         ;counter
            move_string blanks,buffer,26 ;see chapter end
            move_string reply[2],buffer,bx ;see chapter end
            write_ran fcb                ;THIS FUNCTION
            jmp      get_char            ;get another character
all_done:   close    fcb                 ;see Function 10H
```

File Size (Function 23H)



Call
AH = 23H
DS:DX
  Unopened FCB

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

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

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

If no matching directory is found, AL returns FFH.

NOTE

If the value of the Record Size field of the FCB (offset 0EH) is not an even divisor of the file size, the value set in the relative record field is rounded up which yields a size larger than the true file size.

Macro Definition: file_size

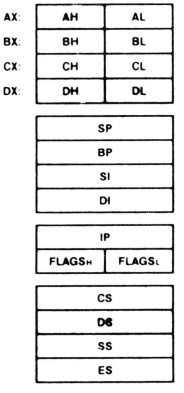
```
macro   fcb
mov     dx,offset fcb
mov     ah,23H
int     21H
endm
```

Example

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

```
fcb             db      37 dup  (?)
prompt          db     "File name: $"
msg1            db     "Record length:     ",0DH,0AH,"$"
msg2            db     "Records:       ",0DH,0AH,"$"
crlf            db      0DH,0AH,"$"
reply           db      17 dup(?)
sixteen         db      10H
;
begin:          display prompt                  ;see Function 09H
                get_string 17,reply             ;see Function 0AH
                cmp     reply[1],0              ;just a CR?
                jne     get_length              ;no, keep going
                jmp     all_done                ;yes, go home
get_length:     display crlf                    ;see Function 09H
                parse   reply[2],fcb            ;see Function 29H
                open    fcb                     ;see Function 0FH
                file_size fcb                   ;THIS FUNCTION
                mov     si,21H                  ;offset to Relative
                                                ;Record field
                mov     di,09H                  ;reply in msg_2
convert_it:     cmp     fcb[si],0               ;digit to convert?
                je      show_it                 ;no, prepare message
                convert fcb[si],ten,msg_2[di]
                inc     si                      ;bump n-o-r index
                inc     di                      ;bump message index
                jmp     convert_it              ;check for a digit
show_it:        convert fcb[0EH],sixteen,msg_1[0FH]
                display msg1                    ;see Function 09H
                display msg2                    ;see Function 09H
                jmp     begin                   ;get a filename
all_done:       close   fcb                     ;see Function 10H
```

Set Relative Record (Function 24H)



Call
AH = 24H
DS:DX
  Opened FCB

Return
None

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

Macro Definition:

```
set_relative_record  macro  fcb
                     mov    dx,offset fcb
                     mov    ah,24H
                     call   loc_50H
                     endm
```

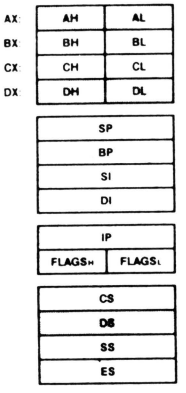
Example

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

```
current_record  equ   20H          ;offset of Current Record
                                   ;field of FCB
file_size       equ   10H          ;offset of File Size
;                                  field of FCB
fcb       db      37 dup (?)
filename  db      17 dup(?)
prompt1   db     "File to copy: $"  ;see Function 09H for
prompt2   db     "Name of copy: $"  ;explanation of $
crlf      db      0DH,0AH,"$"
file_length  dw   ?
buffer    db      32767 dup(?)
```

```
;
begin:    set_dta  buffer                ;see Function 1AH
          display  prompt1               ;see Function 09H
          get_string  15,filename        ;see Function 0AH
          display  crlf                  ;see Function 09H
          parse    filename[2],fcb       ;see Function 29H
          open     fcb                   ;see Function 0FH
          mov      fcb[current_record],0 ;set Current Record
                                         ;field
          set_relative_record  fcb       ;THIS FUNCTION
          mov      ax,word ptr fcb[file_size] ;get file size
          mov      file_length,ax        ;save it for
                                         ;ran_block_write
          ran_block_read  fcb,1,ax       ;see Function 27H
          display  prompt2               ;see Function 09H
          get_string  15,filename        ;see Function 0AH
          display  crlf                  ;see Function 09H
          parse    filename[2],fcb       ;see Function 29H
          create   fcb                   ;see Function 16H
          mov      fcb[current_record],0 ;set Current Record
                                         ;field
          set_relative_record  fcb       ;THIS FUNCTION
          mov      ax,file_length        ;get original file
                                         ;length
          ran_block_write  fcb,1,ax      ;see Function 28H
          close    fcb                   ;see Function 10H
```

Set Vector (Function 25H)



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

Return
None

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

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

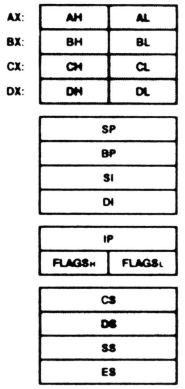
Macro Definition:

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

Example

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

Random Block Read (Function 27H)



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

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

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

AL returns a code that describes the processing.

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

CX returns the number of records read, including the partial record if the AL code is 3; the Current Block (offset 0CH), Current Record (offset 20H), and Relative Record (offset 21H) fields are set to address the next record.

Macro Definition:

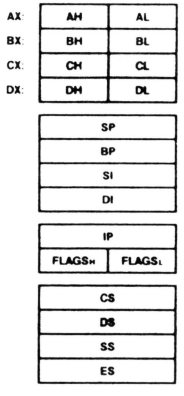
```
ran_block_read  macro  fcb,count,rec_size
                mov    dx,offset fcb
                mov    cx,count
                mov    word ptr fcb[14],rec_size
                mov    ah,27H
                call   loc_50H
                endm
```

Example

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

```
current_record  equ  20H  ;offset of Current Record field
file_size       equ  1OH  ;offset of File Size field
;
fcb       db      37 dup (?)
filename  db      17 dup(?)
prompt1   db     "File to copy: $"   ;see Function 09H for
prompt2   db     "Name of copy: $"   ;explanation of $
crlf      db      0DH,0AH,"$"
file_length  dw   ?
buffer    db      32767 dup(?)
;
begin:    set_dta     buffer           ;see Function 1AH
          display     prompt1          ;see Function 09H
          get_string  15,filename      ;see Function 0AH
          display     crlf             ;see Function 09H
          parse       filename[2],fcb  ;see Function 29H
          open        fcb              ;see Function 0FH
          mov         fcb[current_record],0  ;set Current
                                       ;Record field
          set_relative_record fcb      ;see Function 24H
          mov         ax, word ptr fcb[file_size]
                                       ;get file size
          mov         file_length,ax   ;save it
          ran_block_read  fcb,1,ax     ;THIS FUNCTION
          display     prompt2          ;see Function 09H
          get_string  15,filename      ;see Function 0AH
          display     crlf             ;see Function 09H
          parse       filename[2],fcb  ;see Function 29H
          create      fcb              ;see Function 16H
          mov         fcb[current_record],0;set current
                                       ;Record field
          set_relative_record fcb      ;see Function 24H
          ran_block_write  fcb,1,ax    ;see Function 28H
          close       fcb              ;see Function 1OH
```

Random Block Write (Function 28H)



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

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

DX must contain the offset (to the segment address in DS) of an opened FCB; CX must contain either the number of records to write or 0. The specified number of records (calculated from the Record Size field, offset 0EH) is written from the Disk Transfer Address. The records are written to the file starting at the record specified in the Relative Record field (offset 21H) of the FCB. If CX is 0, no records are written, but the File Size field of the directory entry (offset 1CH) is set to the number of records specified by the Relative Record field of the FCB (offset 21H); allocation units are allocated or released, as required.

AL returns a code that describes the processing:

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

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

Macro Definition:

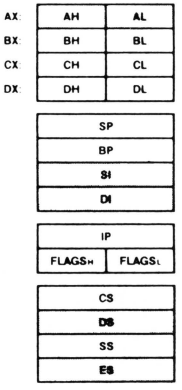
```
ran_block_write  macro  fcb,count,rec_size
                 mov    dx,offset fcb
                 mov    cx,count
                 mov    word ptr fcb[14],rec_size
                 mov    ah,28H
                 call   loc_50H
                 endm
```

Example

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

```
current_record  equ  20H   ;offset of Current Record field
file_size       equ  10H   ;offset of File Size field
;
fcb       db      37 dup (?)
filename  db      17 dup(?)
prompt1   db     "File to copy: $"    ;see Function 09H for
prompt2   db     "Name of copy: $"    ;explanation of $
crlf      db      0DH,0AH,"$"
num_recs  dw      ?
buffer    db      32767 dup(?)
;
begin:    set_dta     buffer       ;see Function 1AH
          display     prompt1      ;see Function 09H
          get_string  15,filename  ;see Function 0AH
          display     crlf         ;see Function 09H
          parse       filename[2],fcb  ;see Function 29H
          open        fcb              ;see Function 0FH
          mov         fcb[current_record],0;set Current
                                           Record field
          set_relative_record fcb      ;see Function 24H
          mov         ax, word ptr fcb[file_size]
                                       ;get file size
          mov         num_recs,ax      ;save it
          ran_block_read  fcb,num_recs,1  ;THIS FUNCTION
          display     prompt2          ;see Function 09H
          get_string  15,filename      ;see Function 0AH
          display     crlf             ;see Function 09H
          parse       filename[2],fcb  ;see Function 29H
          create      fcb              ;see Function 16H
          mov         fcb[current_record],0  ;set Current
                                             ;Record field
          set_relative_record fcb      ;see Function 24H
          ran_block_write fcb,num_recs,1 ;see Function 28H
          close       fcb              ;see Function 10H
```

Parse File Name (Function 29H)



Call
AH = 29H
AL
  Controls parsing (see text)
DS:SI
  String to parse
ES:DI
  Unopened FCB

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

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

Bits 0-3 of AL control the parsing and processing. Bits 4-7 must be set to 0.

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

3 1 The extension in the FCB is not changed if the string does not contain an extension.

0 The extension in the FCB is set to 3 blanks if the string does not contain an extension.

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

Filename separators:

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

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

If the string contains a valid filename:

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

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

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

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

Macro Definition:

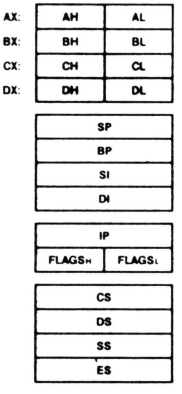
```
parse macro string,fcb
        mov   si,offset string
        mov   di,offset fcb
        push  es
        push  ds
        pop   es
        mov   al,0FH  ;bits 0, 1, 2, 3 on
        mov   ah,29H
        call  loc_50H
        pop   es
        endm
```

Example

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

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

Get Date (Function 2AH)



Call
AH = 2AH

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

This function returns the current date set in the operating system as binary numbers in CX and DX.

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

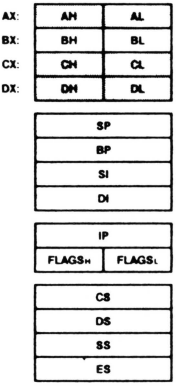
Macro Definition:
```
get_date  macro
          mov   ah,2AH
          call  loc_50H
          endm
```

Example

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

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

Set Date (Function 2BH)



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

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

Registers CX and DX must contain a valid date in binary:

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

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

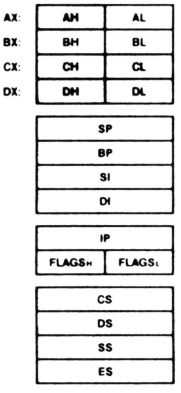
Macro Definition:
```
set_date  macro  year,month,day
          mov    cx,year
          mov    dh,month
          mov    dl,day
          mov    ah,2BH
          call   loc_50H
          endm
```

Example

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

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

Get Time (Function 2CH)



Call
AH = 2CH

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

This function returns the current time set in the operating system as binary numbers in CX and DX.

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

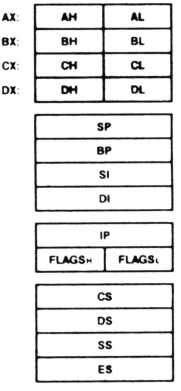
Macro Definition:
```
get_time  macro
          mov   ah,2CH
          call  loc_50H
          endm
```

Example

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

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

Set Time (Function 2DH)



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

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

Registers CX and DX must contain a valid time in binary.

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

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

Macro Definition:

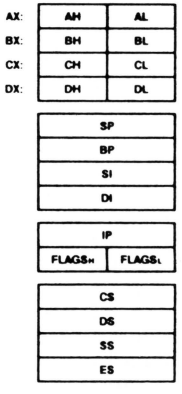
```
set_time  macro  hour,minutes,seconds,hundredths
          mov    ch,hour
          mov    cl,minutes
          mov    dh,seconds
          mov    dl,hundredths
          mov    ah,2DH
          call   loc_50H
          endm
```

Example

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

```
time          db  "00:00:00.00",0DH,0AH,"$"
ten           db   10
;
begin:        set_time  0,0,0,0        ;THIS FUNCTION
read_clock:   get_time                 ;see Function 2CH
              convert  ch,ten,time     ;see end of chapter
              convert  cl,ten,time[3]  ;see end of chapter
              convert  dh,ten,time[6]  ;see end of chapter
              convert  dl,ten,time[9]  ;see end of chapter
              display  time            ;see Function 09H
              dir_console_io 0FFH      ;see Function 06H
              cmp      al,00H          ;was a char. typed?
              jne      stop            ;yes, stop the timer
              jmp      read_clock      ;no keep timer on
stop:         read_kbd                 ;see Function 08H
              jmp      begin           ;keep displaying time
```

Set/Reset Verify Flag (Function 2EH)



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

Return
None

AL must be either 1 (verify after each disk write) or 0 (write without verifying). MS-DOS checks this flag each time it writes to a disk.

The flag is normally off; you may wish to turn it on when writing critical data to disk. Because disk errors are rare and verification slows writing, you will probably want to leave it off at other times.

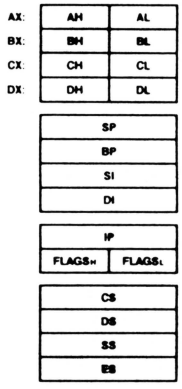
Macro Definition:
```
verify  macro  switch
        mov    al,switch
        mov    ah,2EH
        call   loc_50H
        endm
```

Example

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

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

Get DTA (Function 2FH)



Call
AH = 2FH

Return
ES:BX
Points to Disk Transfer Address

ES returns the segment address of the current Disk Transfer Address; BX returns the offset.

Macro Definition: get_dta

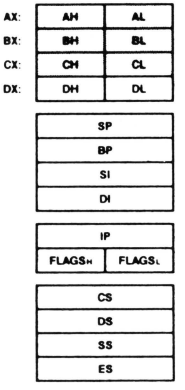
```
macro
mov     ah,2fH
call    loc_50H
endm
```

Example

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

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

Get DOS Version Number (Function 30H)



Call
AH = 30H

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

AL returns the major version number; AH returns the minor version number. (For example, MS-DOS 2.01 returns 2 in AL and 1 in AH.

If AL returns 0, the version of MS-DOS is earlier than 1.28.

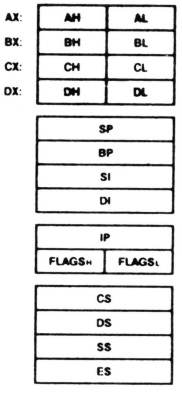
Macro Definition:
```
get_version  macro
             mov    ah,30H
             call   loc_50H
             endm
```

Example

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

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

Terminate but Stay Resident (Function 31H)



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

Return
None

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

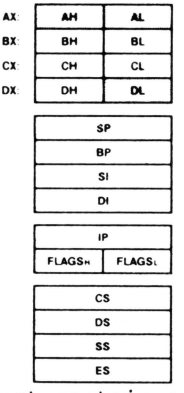
The current process is terminated and MS-DOS tries to set the initial allocation block to the number of paragraphs in DX. No other allocation blocks belonging to the process are released.

The exit code in AL can be retrieved by the parent process with the Wait system call (Function 4DH) and can be tested with the error level batch command.

This Function Request allows more than 64K bytes to remain resident; you should use it, rather than Terminate But Stay Resident (Interrupt 27H), unless your program must be compatible with earlier versions of MS-DOS. This call does not require CS:0 to point to your PSP like INT 27H.

No macro definition or example is included for Function 31H.

CONTROL-C Check (Function 33H)



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

Return
DL
00H = Off
01H = On

AL must contain a code that specifies the requested action:

0 Return current state of CONTROL-C checking.
1 Set state of CONTROL-C checking.

If AL is 1, the value in DL specifies the state to be set (0 = off, 1 = on). If AL is 0, DL returns a code that specifies the current state (0 = off, 1 = on).

If the code in AL is neither 0 nor 1, AL returns FFH and the state of CONTROL-C checking is not affected.

MS-DOS normally checks for CONTROL-C on the controlling device only when performing certain function requests in the 01H through 0CH group. (See specific calls for details). This function request expands CONTROL-C checking to include any function request. For example, with CONTROL-C checking off, all disk I/O proceeds without interruption; with CONTROL-C checking on, the CONTROL-C interrupt is given at the function request that initiates the disk operation.

NOTE

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

Macro Definition:

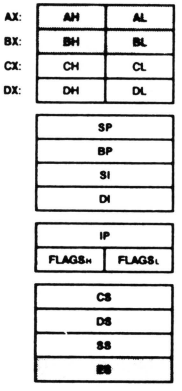
```
ctrl_c_ck  macro  action,state
           mov    al,action
           mov    dl,state
           mov    ah,33H
           call   loc_50H
           endm
```

Example

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

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

Get Vector (Function 35H)



Call
AH = 35H
AL
Interrupt number

Return
ES:BX
Pointer to interrupt routine

AL must contain the number of an interrupt.

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

Your program must use this function request, rather than reading the address directly from the vector table, if the MS-DOS version is 2.00 or greater. If your program must be compatible with earlier versions of MS-DOS, use Function Request 30H (Get DOS Version) to determine which version of MS-DOS is running to choose the correct method.

Macro Definition:

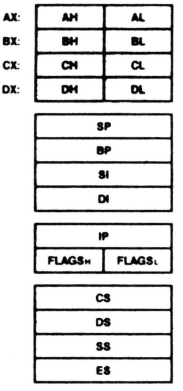
```
get_vector  macro  interrupt
            mov    al,interrupt
            mov    ah,35H
            call   loc_50H
            endm
```

Example

The following program displays the Segment:Offset (CS:IP) for the handler for Interrupt 25H (Absolute Disk Read).

```
message   db     "Interrupt 25H -- CS:0000 IP:0000"
          db      0DH,0AH,"$"
sixteen   db      10H
vec_seg   db      2 dup (?)
vec_off   db      2 dup (?)
;
begin:    get_vector 25H                ;THIS FUNCTION
          mov     word ptr vec_seg,es   ;To access each byte
          mov     word ptr vec_off,bx   ;To access each byte
; SEE END OF CHAPTER FOR DESCRIPTION OF CONVERT MACRO
          convert vec_seg[1],sixteen,message[14h]
          convert vec_seg,sixteen,message[16h]
          convert vec_off[1],sixteen,message[1Ch]
          convert vec_off,sixteen,message[1Eh]
          display message               ;See Function 9
```

Get Disk Free Space (Function 36H)



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

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

DL must contain a drive number
(0 = default, 1 = A, etc.).

If the drive number is valid, the information is returned in the following registers:

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

If the drive number is invalid, AX returns FFFFH.

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

Macro Definition: get_disk_space

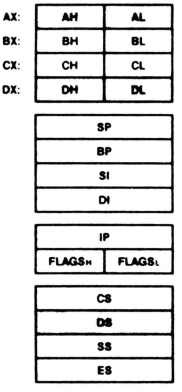
```
macro   drive
mov     dl,drive
mov     ah,36H
call    loc_50H
endm
```

Example

The following program displays (in hexadecimal) the number of bytes available on the disk in drive B.

```
message   db      8 dup (" "),"H "
          db     "bytes available on drive B",0DH,0AH,"$"
sixteen   db      10H
;
begin:    get_disk_space 2                ;THIS FUNCTION
          cmp  ax,ffffh                   ;Error?
          je   error_bad_drive            ;Routine not shown
          mul     cx ;Sectors per cluster * bytes per sector
          mul     bx ;Bytes per cluster * available clusters
          convert dh,sixteen,messagel      ;See end of
          convert dl,sixteen,messagel[02H] ; chapter for
          convert ah,sixteen,messagel[04H] ; description of
          convert al,sixteen,messagel[06H] ; CONVERT macro
          display message                 ;See Function 09H
```

Get Country Data (Function 38H)



Call
AH = 38H
DS:DX
Pointer to 32-byte memory area or DX=-1 to Set Current Country
AL
Country code. In MS-DOS 2.0, must be 0.

Return
Carry set:
AX
2 = File not found
Carry not set:
DS:DX
Pointer to memory area filled with country data

DX must contain either the offset (from the segment address in DS) of a 32-byte memory area in which the country data is to be returned, or -1 to set the country code. AL must contain either a country code or 0 (for current country). The country code is usually the international telephone prefix code.

The following table shows the format of the data returned in the memory area.

Field	Bytes	Offset Hex	Offset Decimal
Date and time format	2 (word)	00	0
Currency symbol	5 (ASCIZ string)	02	2
Thousands separator	2 (ASCIZ string)	07	7
Decimal separator	2 (ASCIZ string)	09	9
Date separator	2 (ASCIZ string)	0B	11
Time separator	2 (ASCIZ string)	0D	13
Bit field	1	0F	15
Currency places	1	10	16
Time format	1	11	17
Case-mapping cell	4 (dword)	12	18
Data-list separator	2 (ASCIZ string)	16	22

Most entries are formatted as an ASCIZ string (ASCII characters terminated by a byte of 00H), but a fixed size is allocated for each field for easy indexing into the table.

```
Date and Time Format: 0 = USA     (h:m:s  m/d/y)
                      1 = Europe  (h:m:s  d/m/y)
                      2 = Japan   (h:m:s  d/m/y)

Bit Field: Bit 1 = 0  No space between symbol and amount
                   1  Space between symbol and amount

           Bit 0 = 0  Currency symbol precedes amount
                   1  Currency symbol follows amount
```

All other bits are undefined.

```
Time format: 0 = 12-hour time
             1 = 24-hour time
```

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

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

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

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

NOTE

This call is fully implemented in MS-DOS 2.01 and higher versions. It exists in version 2.00, but is not fully implemented.

Macro Definition:

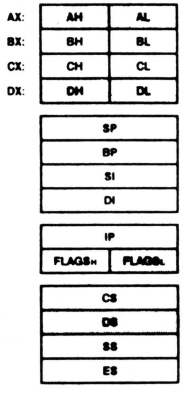
```
get_country macro  mem_area,action
            mov    dx,offset mem_area
            mov    al,action
            mov    ah,38H
            call   loc_50H
            endm
```

Example

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

```
time        db      "  :  :  ",5 dup (20H),"$"
date        db      "  /  /  ",5 dup (20H),"$"
number      db      "999?999?99",0DH,0AH,"$"
ten         db       0AH
data_area   db       32 dup (?)
;
begin:     get_country  data_area,0        ;THIS FUNCTION
           get_time                        ;See Function 2CH
           convert  ch,ten,time            ;See end of chapter
           convert  cl,ten,time[03H]       ;for description of
           convert  dh,ten,time[06H]       ;CONVERT macro
           get_date                        ;See Function 2AH
           sub      cx,1900                ;Want last 2 digits
           convert  cl,ten,date[06H]       ;See end of chapter
           cmp      word ptr data_area,0   ;Check country code
           jne      not_usa                ;It's not USA
           convert  dh,ten,date            ;See end of chapter
           convert  dl,ten,date[03H]       ;See end of chapter
           jmp      all_done               ;Display data
not_usa:   convert  dl,ten,date            ;See end of chapter
           convert  dh,ten,date[03H]       ;See end of chapter
all_done:  mov      al,data_area[04H]      ;Thousand separator
           mov      number[03H],al         ;Put in NUMBER
           mov      al,data_area[06H]      ;Decimal separator
           mov      number[07H],al         ;Put in AMOUNT
           display  time                   ;See Function 09H
           display  date                   ;See Function 09H
           display_char  data_area[02H]    ;See Function 02H
           display  number                 ;See Function 09H
```

Create Directory (Function 39H)



Call
AH = 39H
DS:DX
Pointer to pathname

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

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

If the name is valid, a new directory is created.

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

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

Macro Definition: make_dir

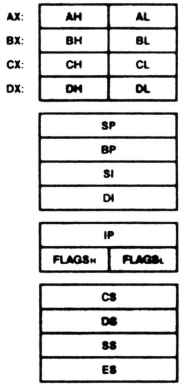
```
macro path
mov     dx,offset path
mov     ah,39H
call    loc_50H
endm
```

Example

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

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

Remove Directory (Function 3AH)



Call
AH = 3AH
DS:DX
Pointer to pathname

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

DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the directory to be erased.

If the name is a valid directory, and the directory is empty except for the "." and ".." entries, the directory is erased.

You cannot erase the current directory. If there is an error, the carry flag (CF) is set and the error code is returned in AX.

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

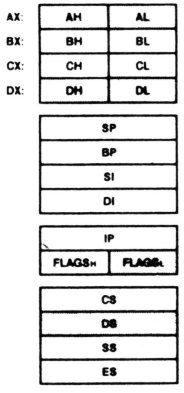
Macro Definition:
```
rem_dir  macro  path
         mov    dx,offset path
         mov    ah,3AH
         call   loc_50H
         endm
```

Example

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

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

Change Current Directory (Function 3BH)



Call
AH = 3BH
DS:DX
Pointer to pathname

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

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

If the name is valid, the current directory is changed to the specified subdirectory.

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

Code	Meaning
3	The path specifies either a file (not a directory) or an invalid path

Macro Definition:

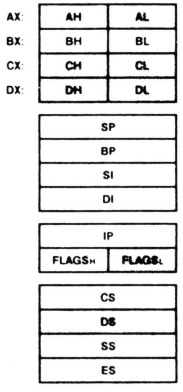
```
change_dir  macro  path
            mov    dx,offset path
            mov    ah,3BH
            call   loc_50H
            endm
```

Example

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

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

Create Handle (Function 3CH)



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

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

DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the file to be created. CX must contain the attribute assigned to the file, as described in Table 1.6.

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

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

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

Macro Definition:

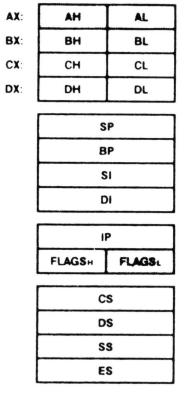
```
create_handle  macro  path,attrib
               mov    dx,offset path
               mov    cx,attrib
               mov    ah,3CH
               call   loc_50H
               endm
```

Example

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

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

Open Handle (Function 3DH)



Call
AH = 3DH
AL = Access code
0 = Read
1 = Write
2 = Both read and write
DS:DX
Pointer to pathname

Return
Carry set:
AX
2 = File not found
4 = Too many open files
5 = Access denied
12 = Invalid access
Carry not set:
AX
Handle

DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the file to be opened. AL must contain the access code:

Code	Meaning
00H	Read
01H	Write
02H	Read and write

AX returns the file handle and the read/write pointer is set to the first byte.

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

Code	Meaning
2	File not found
4	Too many open files (no handle available)
5	Path specifies a volume-id, directory, or read-only file
12	AL is not 00H, 01H, or 02H

The read/write pointer can be changed with Function Request 42H (Move Pointer). The file's date and time can be obtained or set with Function Request 57H (Get/Set Date and Time). The file's attribute can be obtained or changed with Function Request 43 (Change Mode).

Macro Definition:

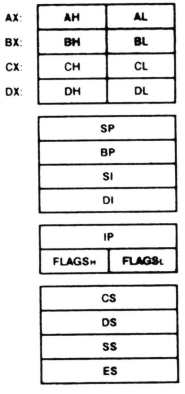
```
open_handle  macro  path,access
             mov    dx,offset path
             mov    al,access
             mov    ah,3DH
             call   loc_50H
             endm
```

Example

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

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

Close File Handle (Function 3EH)



Call
AH = 3EH
BX
File handle

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

BX must contain the handle of an open file.

The file is closed and all internal buffers are flushed.

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

Code	Meaning
6	Handle is not open or is invalid

Macro Definition:

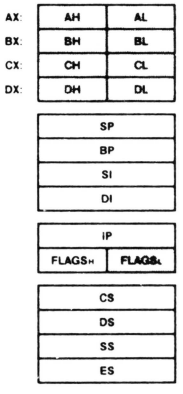
```
close_handle  macro  handle
              mov    bx,handle
              mov    ah,3EH
              call   loc_50H
              endm
```

Example

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

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

Read File or Device (Function 3FH)



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

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

BX must contain the file handle. CX must contain the number of bytes to be read. DX must contain the offset (to the segment address in DS) of the buffer.

AX returns the number of bytes read if carry is not set; if you attempt to read starting at end of file, AX returns 0. The number of bytes specified in CX is not necessarily transferred to the buffer; if you use this call to read from the keyboard, for example, it reads only up to the first CR.

If you use this Function Request to read from standard input, the input can be redirected.

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

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

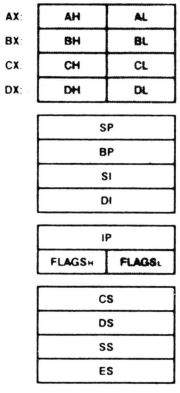
Macro Definition:
```
read_handle  macro  handle,buffer,bytes
             mov    bx,cs:handle
             mov    dx,offset buffer
             mov    cx,cs:bytes
             mov    ah,3FH
             call   cs:loc_50H
             endm
```

Example

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

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

Write File or Device (Function 40H)



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

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

BX must contain the file handle. CX must contain the number of bytes to be written. DX must contain the offset (to the segment address in DS) of the data to be written.

AX returns the number of bytes written if carry is not set. If this value is not the same as you specified in CX, it may indicate an error even though the carry flag isn't set; the most likely reason is a full disk.

If you use this call to write to standard output, the output can be redirected. If you call with CX = 0, the file size is set to the value of the read/write pointer.

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

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

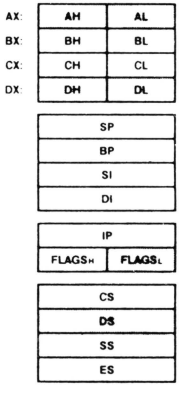
Macro Definition:
```
write_file  macro  handle,data,bytes
            mov    bx,handle
            mov    dx,offset data
            mov    cx,bytes
            mov    ah,40H
            call   loc_50H
            endm
```

Example

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

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

Delete Entry (Function 41H)



Call
AH = 41H
DS:DX
Pointer to pathname

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

DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of the file to be deleted. Wild card characters ("*" and "?") cannot be used.

If the file exists and is not read-only, it is deleted.

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

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

To delete a file with the read-only attribute, first change its attribute to 0 with Function Request 43H (Change Mode).

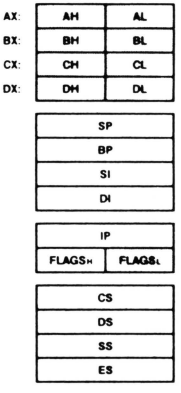
Macro Definition:
```
delete_entry  macro  path
              mov    dx,offset path
              mov    ah,41H
              call   loc_50H
              endm
```

Example

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

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

Move Pointer (Function 42H)



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

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

BX must contain the file handle. CX and DX must contain a 32-bit offset (CX contains the most significant byte). AL must contain a code that specifies how to move the pointer.

Code	Cursor Is Moved To
00H	Beginning of file plus the offset
01H	Current Pointer location plus the offset
02H	End of file plus the offset

DX and AX return the new location of the read/write pointer (a 32-bit integer; DX contains the most significant byte). You can determine the length of a file by setting CX:DX to 0 and specifying 02H in AL; DX:AX return the offset of the byte after the last byte in the file (size of the file in bytes).

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

Code	Meaning
1	AL isn't 00H, 01H, or 02H
6	Handle isn't open

Macro Definition:

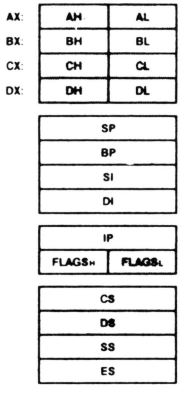
```
move_ptr  macro  handle,high,low,method
          mov    bx,handle
          mov    cx,high
          mov    dx,low
          mov    al,method
          mov    ah,42H
          call   loc_50H
          endm
```

Example

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

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

Change Mode (Function 43H)



Call
AH = 43H
AL = Function
  00H = Get attributes
  01H = Set attributes
CX (if AL = 01H)
  Attribute to be set
DS:DX
  Pointer to pathname

Return
Carry set:
AX
  1 = Invalid function
  3 = Path not found
  5 = Access denied
Carry not set:
CX
  Attributes (if AL = 00H)

DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the pathname of a file. AL must specify whether to get or set the attribute (0 = get, 1 = set). If AL = 1, CX must contain an attribute as described in Table 1.6.

If AL is 0, CX returns the current attribute. If AL is 1, the attribute in CX is set.

Note that you cannot change the volume-ID bit (08H) or the directory bit (10H) with this Function Request.

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

Code	Meaning
1	AL isn't 00H or 01H
3	Path is invalid or file doesn't exist
5	Attribute in CX cannot be changed (directory or volume-id)

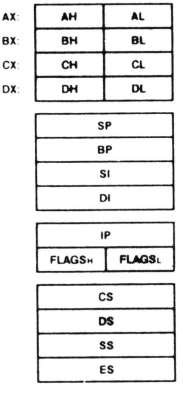
Macro Definition:
```
change_mode  macro  path,action,attrib
             mov    dx,offset path
             mov    al,action
             mov    cx,attrib
             mov    ah,43H
             call   loc_50H
             endm
```

Example

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

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

I/O Control for Devices (Function 44H)



Call

```
AH = 44H
AL
   Function code
BL (if AL = 4 or AL = 5)
   Drive (0 = default, 1 = A, ...)
BX (if AL=0, 1, 2, 3, 6, or 7)
   Handle
CX (if AL = 2, 3, 4, or 5)
   Bytes to read or write
DS:DX (if AL = 2, 3, 4, or 5)
   Pointer to data or buffer
DX (if AL = 1)
   Data
```

Return

```
Carry set:
AX
   1 = Invalid function
   5 = Access denied
   6 = Invalid handle
  13 = Invalid data
Carry not set:
   If AL = 0:
   DX
      Data
   If AL = 2, 3, 4, or 5:
   AX
      Bytes transferred
   If AL = 6 or 7
   AL
      00H = Not ready
      FFH = Ready
```

Either BX must contain the file handle or BL must contain the drive number. AL must contain a code that specifies the function to be performed.

Code	Function
00H	Get device information
01H	Set device information
02H,04H	Read from device control channel
03H,05H	Write to device control channel
06H	Get input status
07H	Get output status

Only codes 00H, 06H, and 07H can be used with disk files; all codes can be used with devices.

Codes 00H and 01H

Code 00H returns device information; code 01H sets device information. If the function code in AL is 00H, DL must contain 00H. The device information is specified or returned in DX; the bits of DX are defined as follows.

If Bit 7 = 1, the channel is a device; other bits have the following meanings:

Bit	Value	Meaning
15		RESERVED.
14	1	Device can process control strings if code is 02H or 03H. This bit can only be read; it cannot be set.
6	0	End of file on input.
5	1	Don't check for control characters.
	0	Check for control characters.
4	1	RESERVED.
3	1	Clock device.
2	1	Null device.
1	1	Console output device.
0	1	Console input device.

NOTE

The control characters referred to in the description of Bit 5 include CONTROL-C, CONTROL-P, CONTROL-S, and CONTROL-Z. To read these characters as data, rather than having them interpreted as control characters, Bit 5 must be set and CONTROL-C checking must be turned off, either with Function Request 33H or the Break command.

If Bit 7 = 0, the channel is a disk file; other bits have the following meanings:

Bit	Value	Meaning
15-8		RESERVED
6	0	Channel has been written
0-5		Block device number of the channel (0 = A, 1 = B, etc.).

Codes 02H, 03H, 04H, and 05H

Codes 02H and 03H send or receive control data to or from a character device (printer, serial port, etc.); codes 04H and 05H send or receive control data to or from a block device.

DS must contain the segment address of the area of memory to which the data is to be read or from which it is to be written; DX must contain the offset. CX must contain the number of bytes to be read or written. For codes 02H and 03H, BX must contain a handle; for codes 04H and 05H, BL must contain a drive number (0 = default, 1 = A, etc.).

AX returns the number of bytes transferred.

Codes 06H and 07H

These codes check whether the file or device associated with a handle is ready for input (06H) or output (07H).

BX must contain the handle.
The status is returned in AL:

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

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

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

Code	Meaning
1	DH is not 0 (code 01H) AL is not 00H-07H, or the device cannot perform the specified function (codes 02H, 03H, 04H, 05H)
5	Drive in BL is invalid (codes 04H and 05H) Access denied (codes 06H and 07H)
6	Handle is not open (codes 00H-03H, 06H, 07H)
13	Invalid data (codes 06H and 07H)

Macro Definition:

```
io_ctrl macro  code,handle,buffer,attrib
        mov    al,code
        mov    bx,handle
        mov    cx,attrib
        mov    dx,offset buffer
        mov    ah,44H
        call   loc_50H
        endm
```

Example

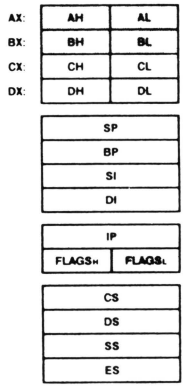
```
      mov    bx,Handle
  (or mov    bl,drive     for codes 04H and 05H
                          0 = default, 1 = A, etc.)

      mov    dx,Data
  (or lds    dx,buf       and
      mov    cx,count     for codes 02H-05H)

      mov    ah,44H
      mov    al,code
      call   loc_50H

; For codes 02H-05H, AX is the number of bytes transferred
; (same as Read and Write).
; For codes 06H and 07H, AL is status returned, AL=0 if
; status is Not Ready, AL=0FFH otherwise.
```

Duplicate a File Handle (Function 45H)



Call
AH = 45H
BX
Handle

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

BX must contain the handle of an open file.

AX returns a new file handle that refers to the same file at the same position.

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

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

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

Macro Definition:

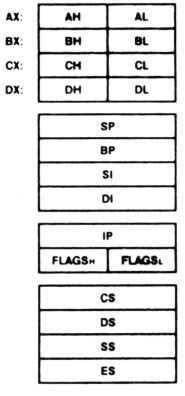
```
xdup  macro  handle
      mov    bx,handle
      mov    ah,45H
      call   loc_50H
      endm
```

Example

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

```
pgm_file    db      "command.com",0
cmd_line    db       9,"/c dir /w",0DH
parm_blk    db       14 dup (0)
path        db      "dirfile",0
new_stdout  db       ?
;
begin:      set_block  last_inst    ;See Function 4AH
            jc       error_setblk   ;Routine not shown
            create_handle  path,0   ;See Function 3CH
            jc       error_create   ;Routine not shown
            mov      dir_file,ax    ;Handle of new file
            xdup     1              ;THIS FUNCTION
            jc       error_xdup     ;Routine not shown
            mov      new_stdout,ax  ;Alternate STDOUT
            xdup2    dir_file,1     ;See Function 46H
            jc       error_xdup2    ;Routine not shown
            exec     pgm_file,cmd_line,parm_blk,0
            jc       error_exec     ;Routine not shown
            xdup2    new_stdout,1   ;See Function 46H
            jc       error_xdup2    ;Routine not shown
            close_handle  new_stdout ;See Function 3EH
            jc       error_close    ;Routine not shown
```

Force Duplicate File Handle (Function 46H)



Call
AH = 46H
BX
Handle
CX
Second handle

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

BX must contain the handle of an open file; CX must contain a second file handle.

If the file referred to by the handle in CX is open, it is closed; on return, the handle in CX refers to the same file at the same position as the handle in AX.

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

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

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

Macro Definition:

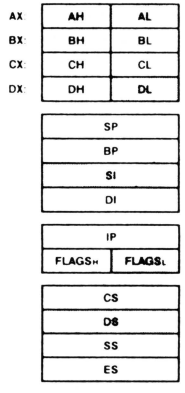
```
xdup2   macro   handle1,handle2
        mov     bx,handle1
        mov     cx,handle2
        mov     ah,46H
        call    loc_50H
        endm
```

Example

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

```
pgm_file    db      "command.com",0
cmd_line    db       9,"/c dir /w",0DH
parm_blk    db       14 dup (0)
path        db      "dirfile",0
new_stdout  db       ?
;
begin:      set_block  last_inst    ;See Function 4AH
            jc       error_setblk   ;Routine not shown
            create_handle  path,0   ;See Function 3CH
            jc       error_create   ;Routine not shown
            mov      dir_file,ax    ;Handle of new file
            xdup     1              ;See Function 45H
            jc       error_xdup     ;Routine not shown
            mov      new_stdout,ax  ;Alternate STDOUT
            xdup2    dir_file,1     ;THIS FUNCTION
            jc       error_xdup2    ;Routine not shown
            exec     pgm_file,cmd_line,parm_blk,0
            jc       error_exec     ;Routine not shown
            xdup2    new_stdout,1   ;THIS FUNCTION
            jc       error_xdup2    ;Routine not shown
            close_handle  new_stdout ;See Function 3E
            jc       error_close    ;Routine not shown
```

Get Current Directory (Function 47H)



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

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

DL must contain a drive number (0 = default, 1 = A, etc.). SI must contain the offset (from the segment address in DS) of a 64-byte memory area.

The call places an ASCIZ string that specifies the full pathname of the current directory for the specified drive, starting from the root directory, in the memory area. The string does not begin with a backslash and does not include the drive letter.

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

Code	Meaning
15	Drive in DL is invalid

Macro Definition:

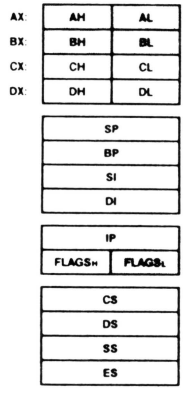
```
get_dir  macro   drive,buffer
         mov     dl,drive
         mov     si,offset buffer
         mov     ah,47H
         call    loc_50H
         endm
```

Example

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

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

Allocate Memory (Function 48H)



Call
AH = 48H
BX
  Paragraphs of memory

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

BX must contain the amount of memory requested, in paragraphs (1 paragraph = 16 bytes).

AX returns the segment address (with an implied offset of 0) of the allocated memory. If the allocation fails, BX returns the maximum number of paragraphs of contiguous memory that could be allocated.

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

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

Macro Definition: allocate_memory

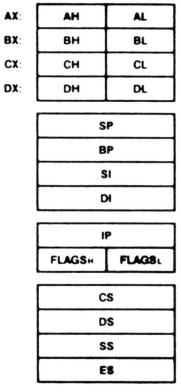
```
macro   bytes
mov     bx,bytes
mov     cl,4
shr     bx,cl
inc     bx
mov     ah,48H
call    loc_50H
endm
```

Example

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

```
path      db      "textfile.asc",0
handle    dw       ?
mem_seg   dw       ?
file_len  dw       ?
;
begin:    open_handle  path,0
          jc       error_open     ;Routine not shown
          mov      handle,ax      ;Save handle
          move_ptr handle,0,0,2   ;See Function 42H
          jc       error_move     ;Routine not shown
          mov      file_len,ax    ;Save file length
          set_block  last_inst    ;See Function 4AH
          jc       error_setblk   ;Routine not shown
          allocate_memory  file_len ;THIS FUNCTION
          jc       error_alloc    ;Routine not shown
          mov      mem_seg,ax     ;Save address of new memory
          mov_ptr  handle,0,0,0   ;See Function 42H
          jc       error_move     ;Routine not shown
          push     ds             ;Save DS
          mov      ax,mem_seg     ;Get segment of new memory
          mov      ds,ax          ;Point DS at new memory
          read_handle  handle,code,file_len ;Read file into
;                                                 new memory
          pop      ds             ;Restore DS
          jc       error_read     ;Routine not shown
;         <CODE TO PROCESS FILE GOES HERE>
          free_memory  mem_seg    ;See Function 49H
          jc       error_freemem  ;Routine not shown
```

Free Allocated Memory (Function 49H)



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

Return
Carry set:
AX
7 = Memory control blocks damaged
9 = Incorrect segment
Carry not set:
No error

ES must contain the segment address of the memory area being returned.

The specified area of memory is returned to the system pool. The memory must have been allocated with Function Request 48H (Allocate Memory).

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

Code	Meaning
7	Memory control blocks damaged (a user program changed memory that doesn't belong to it)
9	The memory pointed to by ES was not allocated with Function Request 48H

Macro Definition:

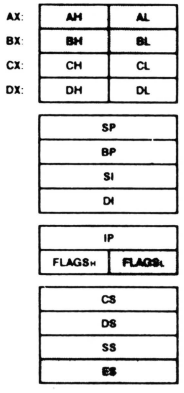
```
free_memory  macro   seg_addr
             mov     ax,seg_addr
             mov     es,ax
             mov     ah,49H
             call    loc_50H
             endm
```

Example

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

```
path      db      "textfile.asc",0
handle    dw       ?
mem_seg   dw       ?
file_len  dw       ?
;
begin:    open_handle  path,0
          jc       error_open    ;Routine not shown
          mov      handle,ax     ;Save handle
          move_ptr handle,0,0,2  ;See Function 42H
          jc       error_move    ;Routine not shown
          mov      file_len,ax   ;Save file length
          set_block  last_inst   ;See Function 4AH
          jc       error_setblk  ;Routine not shown
          allocate_memory  file_len ;See Function 48H
          jc       error_alloc   ;Routine not shown
          mov      mem_seg,ax    ;Save address of new memory
          mov_ptr  handle,0,0,0  ;See Function 42H
          jc       error_move    ;Routine not shown
          push     ds            ;Save DS
          mov      ax,mem_seg    ;Get segment of new memory
          mov      ds,ax         ;Point DS at new memory
          read_handle  handle,code,file_len ;Read file into
                                             new memory
          pop      ds            ;Restore DS
          jc       error_read    ;Routine not shown
          <CODE TO PROCESS FILE GOES HERE>
          free_memory  mem_seg   ;THIS FUNCTION
          jc       error_freemem ;Routine not shown
```

Set Block (Function 4AH)



Call
AH = 4AH
BX
Paragraphs of memory
ES
Segment address of memory area

Return
Carry set:
AX
7 = Memory control blocks damaged
8 = Insufficient memory
9 = Incorrect segment
BX
Paragraphs of memory available
Carry not set:
No error

ES must contain the segment address of the area of memory to be modified (for a program, this is the start of the Program Segment Prefix). BX must contain the new amount of memory requested, in paragraphs (1 paragraph = 16 bytes).

MS-DOS attempts to change the size of the area of memory.

If the call fails on a request to increase memory, BX returns the maximum number of paragraphs of memory to which the program can be increased.

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

Code	Meaning
7	Memory control blocks destroyed (a user program changed memory that doesn't belong to it)
8	Not enough free memory
9	The memory pointed to by ES cannot be allocated with Set Block

Macro Definition:

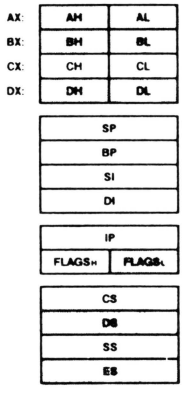
```
set_block  macro   last_byte
           mov     bx,offset last_byte
           mov     cl,4
           shr     bx,cl
           inc     bx
           mov     ah,4AH
           call    loc_50H
           endm
```

Example

The following program invokes a second copy of COMMAND.COM and executes a DIR (directory) command.

```
pgm_file  db       "command.com",0
cmd_line  db        9,"/c dir /w",0DH
parm_blk  db        14 dup (?)
reg_save  db        10 dup (?)
;
begin: set_block  last_inst                    ;THIS FUNCTION
       exec       pgm_file,cmd_line,parm_blk,0 ;See Function
                                                ;4BH
```

Execute Program (Function 4BH)



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

Return
Carry set:
AX
  1 = Invalid function
  2 = File not found
  8 = Insufficient memory
 10 = Bad environment
 11 = Bad format
Carry not set:
  No error

DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies the drive and pathname of an executable program file. BX must contain the offset (from the segment address in ES) of a parameter block. AL must contain a code that specifies whether to execute the program.

Code	Meaning
00H	Load and execute the program.
03H	Load an overlay (or program).

Code 00H -- Load and Execute Program

There must be enough free memory for MS-DOS to load the program file. Usually, all available memory is allocated to the invoking program when it is loaded, so you must free some memory with Set Block (Function Request 4AH) before loading another program with this function request. Unless memory is needed for some other purpose, shrink to the minimum amount of memory the invoking program needs.

A Program Segment Prefix is created for the program being loaded; the terminate and CONTROL-C addresses are set to the instruction that immediately follows the Execute Program Function Request in the invoking program.

The following is the format of the parameter block if the program is to be executed.

Offset (Hex)	Length (Bytes)	Description
00	2 (word)	Segment address of environment to be passed; 00H means copy the parent's environment.
02	4 (dword)	Segment:Offset of command line to be placed at offset 80H of the new Program Segment Prefix. This must be a correctly formed command line no longer than 128 bytes.
06	4 (dword)	Segment:Offset of FCB to be placed at offset 5CH of the new Program Segment Prefix. (See the description of the Program Segment Prefix in Chapter 4.)
0A	4 (dword)	Segment:Offset of FCB to be placed at offset 6CH of the new Program Segment Prefix. (See the description of the Program Segment Prefix in Chapter 4.)

All open files of a program are available to the newly loaded program, giving the parent program control over the definition of standard input, output, auxiliary, and printer devices. For example, a program could write a series of records to a file, open the file as standard input, open a second file as standard output, then use Execute Program to load and execute a program that takes its input from standard input, sorts records, and writes to standard output.

The loaded program also receives an environment, a series of ASCIZ strings of the form parameter=value (for example, VERIFY=ON). The environment must begin on a paragraph boundary, be less than 32K bytes long, and end with a byte of 00H (that is, the final entry consists of an ASCII string followed by two bytes of 00H). The loaded program either inherits a copy of the parent's environment or receives a new environment built for it by the parent.

Place the segment address of the environment at offset 2CH of the new Program Segment Prefix. To build an environment for the loaded program, put it on a paragraph boundary and place the segment address of the environment in the first word of the parameter block. To pass a copy of the parent's environment to the loaded program, put 00H in the first word of the parameter block.

Code 03H -- Load But Don't Execute

MS-DOS assumes that the invoking program is loading into its own address space, so no free memory is required. A Program Segment Prefix is not created. This code is used to load a program overlay.

The format of the parameter block:

Offset (Hex)	Length (Bytes)	Description
00	2 (word)	Segment address where program is to be loaded.
02	2 (word)	Relocation factor (typically, this is the same as the word at 00H; see the description of the .EXE file format in Chapter 5).

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

Code	Meaning
1	AL is not 00H or 03H
2	Program file not found or path is invalid
8	Not enough memory to load the program
11	Program file is an .EXE file that contains internally inconsistent information

Macro Definition:

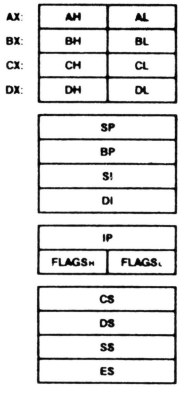
```
exec  macro  path,command,parms,function
      mov    dx,offset path
      mov    bx,offset parms
      mov    word ptr parms[02H],offset command
      mov    word ptr parms[04H],cs
      mov    word ptr parms[06H],5CH
      mov    word ptr parms[08H],es
      mov    word ptr parms[0AH],6CH
      mov    word ptr parms[0CH],es
      mov    al,function
      mov    ah,4BH
      call   loc_50H
      endm
```

Example

The following program invokes a second copy of COMMAND.COM and executes a DIR (directory) command:

```
pgm_file  db      "command.com",0
cmd_line  db       9,"/c dir /w",0DH
parm_blk  db       14 dup (?)
reg_save  db       10 dup (?)
;
begin:
   set_block  last_inst                    ;See Function 4BH
   exec       pgm_file,cmd_line,parm_blk,0 ;THIS FUNCTION
```

End Process (Function 4CH)



Call
AH = 4CH
AL
Return code

Return
None

AL contains a return code.

All open handles are closed, the current process is ended, and control returns to the invoking process. The return code can be retrieved with Wait (Function Request 4DH) and the batch subcommands IF and ERRORLEVEL.

This function request doesn't require that CS contain the segment address of the Program Segment Prefix. You should use it to end a program (rather than Interrupt 20H or a jump to location 0) unless your program must be compatible with earlier versions of MS-DOS.

Macro Definition:

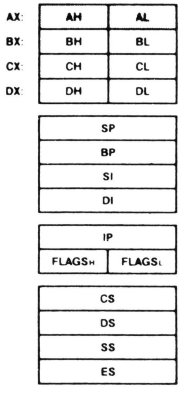
```
end_process  macro  return_code
             mov    al,return_code
             mov    ah,4CH
             call   loc_50H
             endm
```

Example

The following program displays a message and returns to MS-DOS with a return code of 8. It uses only the opening portion of the sample program skeleton shown in Figure 1.2.

```
message   db      "Displayed by FUNC_4CH example",0DH,0AH,"$"
;
begin:    display       message    ;See Function 09H
          end_process   8          ;THIS FUNCTION
code      ends
          end           code
```

Wait (Function 4DH)



Call
AH = 4DH

Return
AX
Return code

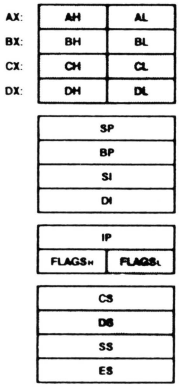
AL returns the return code specified by a child program (with Function Request 31H or 4CH). AH returns a code that specifies the reason the program ended.

Code	Meaning
00H	Normal ending
01H	CONTROL-C
02H	Critical device error
03H	Function 31H (Terminate and Stay Resident)

The exit code is returned only once.

No macro definition or example is included for Function 40H.

Find First File (Function 4EH)



Call
AH = 4EH
DS:DX
Pointer to pathname
CX
Attributes to match

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

DX must contain the offset (from the segment address in DS) of an ASCIZ string that specifies a path and filename; the filename can contain global characters ("*" and "?"). CX must contain the attribute to be used in searching for the file, as described in Table 1.6.

If a directory entry is found that matches the name and attribute, the current DTA is filled as follows:

Offset	Length	Description
00H	21	Reserved for subsequent Find Next File (Function Request 4FH).
15H	1	Attribute found.
16H	2	Time file was last written.
18H	2	Date file was last written.
1AH	2	Low word of file size.
1CH	2	High word of file size.
1EH	13	Name and extension of the file, followed by 00H. All blanks are removed; if there is an extension, it is preceded by a period (it appears just as you would enter it in a command).

NOTE

If the attribute field is set for hidden or system files, or directory entries, it is considered to be 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).

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

Code	Meaning
2	Path is invalid
18	No matching directory entry

Macro Definition: find_first_file

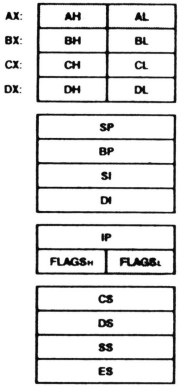
```
macro   path,attrib
mov     dx,offset path
mov     cx,attrib
mov     ah,4EH
call    loc_50H
endm
```

Example

The following program displays a message that specifies whether a file named REPORT.ASM exists in the current directory on the disk in drive B.

```
yes         db      "FILE EXISTS.",0DH,0AH,"$"
no          db      "FILE DOES NOT EXIST.",0DH,0AH,"$"
path        db      "b:report.asm",0
buffer      db       43 dup (?)
;
begin:      set_dta  buffer             ;See Function 1AH
            find_first_file  path,0      ;THIS FUNCTION
            jc       error_findfirst    ;Routine not shown
            cmp      al,12H             ;File found?
            je       not_there          ;No
            display  yes                ;See Function 09H
            jmp      return             ;All done
not_there:  display  no                 ;See Function 09H
```

Find Next File (Function 4FH)



Call
AH = 4FH

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

The current DTA must contain the information filled in by a previous Find First File (Function Request 4EH).

MS-DOS searches for the next directory entry that matches the name and attributes specified in the previous Find First File (Function Request 4EH).

If a matching entry is found, the current DTA is filled just as it was for Find First File.

Offset	Length	Description
00H	21	Reserved for subsequent Find Next File (Function Request 4FH).
15H	1	Attribute found.
16H	2	Time file was last written.
18H	2	Date file was last written.
1AH	2	Low word of file size.
1CH	2	High word of file size.
1EH	13	Name and extension of the file, followed by 00H. All blanks are removed; if there is an extension, it is preceded by a period (it appears just as you would enter it in a command).

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

Code	Meaning
2	Path is invalid
18	No matching directory entry

Macro Definition:

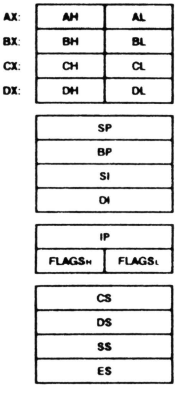
```
find_next_file  macro
                mov     ah,4FH
                call    loc_50H
                endm
```

Example

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

```
message     db      "No files",0DH,0AH,"$"
files       db       ?
ten         db       0AH
path        db      "b:*.*",0
buffer      db       43 dup (?)
;
begin:      set_dta  buffer           ;See Function 1AH
            find_first_file  path,0   ;See Function 4EH
            jc       error_findfirst  ;Routine not shown
            cmp      al,12H           ;Directory empty?
            je       all_done         ;Yes, go home
            inc      files            ;No, bump file counter
search_dir: find_next_file            ;THIS FUNCTION
            jc       error_findnext   ;Routine not shown
            cmp      al,12H           ;Any more entries?
            je       done             ;No, go home
            inc      files            ;Yes, bump file counter
            jmp      search_dir       ;And check again
done:       convert  files,ten,message ;See end of chapter
all_done:   display  message          ;See Function 09H
```

Get/Verify State (Function 54H)



Call
AH = 54H

Return
AL
Current verify status
00H = No verify after write
01H = Verify after write

AL returns 00H if verify is off, 01H if verify is on.

The verify status can be set with Set/Reset Verify Flag, Function Request 2EH.

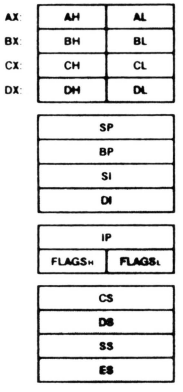
Macro Definition:
```
get_verify  macro
            mov     ah,54H
            call    loc_50H
            endm
```

Example

The following program displays the verify status:

```
message   db      "Verify ","$"
on        db      "on.",0DH,0AH,"$"
off       db      "off.",0DH,0AH,"$"
;
begin:    display message       ;See Function 09H
          get_verify            ;THIS FUNCTION
          cmp     al,0          ;Is flag off?
          jg      ver_on        ;No, it's on
          display off           ;See Function 09H
          jmp     return        ;Go home
ver_on:   display on            ;See Function 09H
```

Rename a File (Function 56H)



Call
AH = 56H
DS:DX
Pointer to pathname
ES:DI
Pointer to second pathname

Return
Carry set:
AX
2 = File not found
5 = Access denied
17 = Not same device

DX must contain the offset (from the segment address in DS) of an ASCIZ string that contains an optional drive and path, and a filename. DI must contain the offset (from the segment address in ES) of an ASCIZ string that contains a second optional drive and path, and a filename.

If the first file exists, it is renamed to the second name.

If the second path specifies a drive, the first path must specify or imply the same drive. The directory paths do not need to be the same, allowing you to move a file to another directory at the same time you rename it.

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

Code	Meaning
2	File not found or path is invalid
5	First pathname specifies a directory; second path specifies an existing file; or the second directory entry could not be opened
17	Files not on the same drive

Macro Definition:

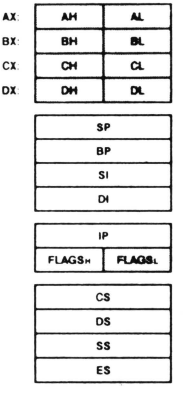
```
rename_file  macro  old_path,new_path
             mov    dx,offset old_pat
             push   ds
             pop    es
             mov    di,offset new_path
             mov    ah,56H
             call   loc_50H
             endm
```

Example

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

```
prompt1   db     "Filename: $"
prompt2   db     "New name: $"
old_path  db      15,?,15 dup (?)
new_path  db      15,?,15 dup (?)
crlf      db      0DH,0AH,"$"
;
begin:    display prompt1              ;See Function 09H
          get_string  15,old_path      ;See Function 0AH
          xor     bx,bx                ;To use BL as index
          mov     bl,old_path[1]       ;Get string length
          mov     old_path[bx+2],0     ;Make an ASCIZ string
          display crlf                 ;See Function 09H
          display prompt2              ;See Function 09H
          get_string  15,new_path      ;See Function 0AH
          xor     bx,bx                ;To use BL as index
          mov     bl,new_path[1]       ;Get string length
          mov     new_path[bx+2],0     ;Make an ASCIZ string
          display crlf                 ;See Function 09H
          rename_file old_path[2],new_path[2];THIS FUNCTION
          jc      error_rename         ;Routine not shown
```

Get/Set Date or Time (Function 57H)



Call
AH = 57H
AL = Function code
  00H = Get date and time
  01H = Set date and time
BX
  Handle
CX (if AL = 01H)
  Time to be set
DX (if AL = 01H)
  Date to be set

Return
Carry set:
AX
  1 = Invalid function
  6 = Invalid handle
Carry not set:
CX (if AL = 00H)
  Time file last written
DX (if AL = 00H)
  Date file last written

AL must contain either 0 to get the date and time or 1 to set the date and time. BX must contain the file handle. If AL=1, CX must contain the time and DX must contain the date, in the form described in Table 1.2.

If AL=0, CX returns the time and DX returns the date, in the same form.

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

Code	Meaning
1	AL is not 00H or 01H
6	Handle is not open

```
      mov     bl,new_path[1]     ;Get string length
```

Macro Definition:

```
get_set_date_time macro  handle,action,time,date
                  mov    bx,handle
                  mov    al,action
                  mov    cx,word ptr time
                  mov    dx,word ptr date
                  mov    ah,57H
                  call   loc_50H
                  endm
```

Example

The following program gets the date of the file named REPORT.ASM in the current directory on the disk in drive B, increments the day, increments the month or year if necessary, and sets the new date of the file.

```
month     db      31,28,31,30,31,30,31,31,30,31,30,31
path      db     "b:report.asm",0
handle    dw      ?
time      db      2 dup (?)
date      db      2 dup (?)
;
begin:    open_handle  path,0          ;See Function 3DH
          mov     handle,ax            ;Save handle
          get_set_date_time handle,0,time,date;THISFUNCTION
          jc      error_time           ;Routine not shown
          mov     word ptr time,cx     ;Save time
          mov     word ptr date,dx     ;Save date
          convert_date  date[-24]      ;See end of chapter
          inc     dh                   ;Increment day
          xor     bx,bx                ;To use BL as index
          mov     bl,dl                ;Get month
          cmp     dh,month[bx-1]       ;Past last day?
          jle     month_ok             ;No, go home
          mov     dh,1                 ;Yes, set day to 1
          inc     dl                   ;Increment month
          cmp     dl,12                ;Is it past December?
          jle     month_ok             ;No, go home
          mov     dl,1                 ;Yes, set month to 1
          inc     cx                   ;Increment year
month_ok: pack_date  date              ;See end of chapter
          get_set_date_time handle,1,time,date;THISFUNCTION
          jc      error_time           ;Routine not shown
          close_handle  handle         ;See Function 3EH
          jc      error_close          ;Routine not shown
```

1.9 MACRO DEFINITIONS FOR MS-DOS SYSTEM CALL EXAMPLES

```
;
;*******************
; Interrupts
;*******************
;                                       INTERRUPT 25H
ABS_DISK_READ  macro  disk,buffer,num_sectors,first_sector
     mov    al,disk
     mov    bx,offset buffer
     mov    cx,num_sectors
     mov    dx,first_sector
     int    25H
     popf
     endm
;                                       INTERRUPT 26H
ABS_DISK_WRITE  macro  disk,buffer,num_sectors,first_sector
     mov    al,disk
     mov    bx,offset buffer
     mov    cx,num_sectors
     mov    dx,first_sector
     int    26H
     popf
     endm
;                                       INTERRUPT 27H
STAY_RESIDENT  macro  last_instruc
     mov    dx,offset last_instruc
     inc    dx
     int    27H
     endm
;
;
;*******************
; Function Requests
;*******************
;                                    FUNCTION REQUEST 00H
TERMINATE_PROGRAM  macro
     xor    ah,ah
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 01H
READ_KBD_AND_ECHO  macro
     mov    ah,01H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 02H
DISPLAY_CHAR  macro  character
     mov    dl,character
     mov    ah,02H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 03H
AUX_INPUT    macro
     mov    ah,03H
     call   loc_50H
```

```
     endm
;                                    FUNCTION REQUEST 04H
AUX_OUTPUT  macro
     mov    ah,04H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 05H
PRINT_CHAR  macro  character
     mov    dl,character
     mov    ah,05H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 06H
DIR_CONSOLE_IO  macro  switch
     mov    dl,switch
     mov    ah,06H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 07H
DIR_CONSOLE_INPUT  macro
     mov    ah,07H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 08H
READ_KBD  macro
     mov    ah,08H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 09H
DISPLAY  macro  string
     mov    dx,offset string
     mov    ah,09H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 0AH
GET_STRING  macro  limit,string
     mov    dx,offset string
     mov    string,limit
     mov    ah,0AH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 0BH
CHECK_KBD_STATUS  macro
     mov    ah,0BH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 0CH
FLUSH_AND_READ_KBD  macro  switch
     mov    al,switch
     mov    ah,0CH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 0DH
RESET_DISK  macro
     mov    ah,0DH
```

```
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 0EH
SELECT_DISK  macro  disk
     mov    dl,disk[-65]
     mov    ah,0EH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 0FH
OPEN  macro  fcb
     mov    dx,offset fcb
     mov    ah,0FH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 10H
CLOSE  macro  fcb
     mov    dx,offset fcb
     mov    ah,10H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 11H
SEARCH_FIRST  macro  fcb
     mov    dx,offset fcb
     mov    ah,11H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 12H
SEARCH_NEXT  macro  fcb
     mov    dx,offset fcb
     mov    ah,12H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 13H
DELETE  macro  fcb
     mov    dx,offset fcb
     mov    ah,13H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 14H
READ_SEQ  macro  fcb
     mov    dx,offset fcb
     mov    ah,14H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 15H
WRITE_SEQ  macro  fcb
     mov    dx,offset fcb
     mov    ah,15H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 16H
CREATE  macro  fcb
     mov    dx,offset fcb
     mov    ah,16H
     call   loc_50H
```

```
     endm
;                                    FUNCTION REQUEST 17H
RENAME  macro  fcb,newname
     mov    dx,offset fcb
     mov    ah,17H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 19H
CURRENT_DISK  macro
     mov    ah,19H
     call   loc_50H
     endm
;                                     FUNCTION REQUEST 1AH
SET_DTA  macro  buffer
     mov    dx,offset buffer
     mov    ah,1AH
     call   loc_50H
     endm
;                                     FUNCTION REQUEST 21H
READ_RAN  macro  fcb
     mov    dx,offset fcb
     mov    ah,21H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 22H
WRITE_RAN  macro  fcb
     mov    dx,offset fcb
     mov    ah,22H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 23H
FILE_SIZE  macro  fcb
     mov    dx,offset fcb
     mov    ah,23H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 24H
SET_RELATIVE_RECORD  macro  fcb
     mov    dx,offset fcb
     mov    ah,24H
     call   loc_50H
     endm
;                                     FUNCTION REQUEST 25H
SET_VECTOR  macro  interrupt,handler_start
     mov    al,interrupt
     mov    dx,offset handler_start
     mov    ah,25H
     call   loc_50H
     endm
;                                     FUNCTION REQUEST 27H
RAN_BLOCK_READ  macro  fcb,count,rec_size
     mov    dx,offset fcb
     mov    cx,count
     mov    word ptr fcb[14],rec_size
     mov    ah,27H
```

```
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 28H
RAN_BLOCK_WRITE  macro  fcb,count,rec_size
     mov    dx,offset fcb
     mov    cx,count
     mov    word ptr fcb[14],rec_size
     mov    ah,28H
     call   loc_50H
     endm
;                                   FUNCTION REQUEST 29H
PARSE  macro  string,fcb
     mov    si,offset string
     mov    di,offset fcb
     push   es
     push   ds
     pop    es
     mov    al,0FH
     mov    ah,29H
     call   loc_50H
     pop    es
     endm
;                                    FUNCTION REQUEST 2AH
GET_DATE  macro
     mov    ah,2AH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 2BH
SET_DATE  macro  year,month,day
     mov    cx,year
     mov    dh,month
     mov    dl,day
     mov    ah,2BH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 2CH
GET_TIME  macro
     mov    ah,2CH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 2DH
SET_TIME  macro  hour,minutes,seconds,hundredths
     mov    ch,hour
     mov    cl,minutes
     mov    dh,seconds
     mov    dl,hundredths
     mov    ah,2DH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 2EH
VERIFY  macro  switch
     mov    al,switch
     mov    ah,2EH
     call   loc_50H
     endm
```

```
;                                       FUNCTION REQUEST 2FH
GET_DTA  macro
     mov    ah,2FH
     call   loc_50H
     endm
;                                       FUNCTION REQUEST 30H
GET_VERSION  macro
     mov    ah,30H
     call   loc_50H
     endm
;                                       FUNCTION REQUEST 33H
CTRL_C_CK  macro  action,state
     mov    al,action
     mov    dl,state
     mov    ah,33H
     call   loc_50H
     endm
;                                       FUNCTION REQUEST 35H
GET_VECTOR  macro  interrupt
     mov    al,interrupt
     mov    ah,35H
     call   loc_50H
     endm
;                                       FUNCTION REQUEST 36H
GET_DISK_SPACE  macro  drive
     mov    dl,drive
     mov    ah,36H
     call   loc_50H
     endm
;                                  ; FUNCTION REQUEST 38H
GET_COUNTRY  macro  mem_area,action
     mov    dx,offset mem_area
     mov    al,action
     mov    ah,38H
     call   loc_50H
     endm
;                                        FUNCTION REQUEST 39H
MAKE_DIR  macro  path
     mov    dx,offset path
     mov    ah,39H
     call   loc_50H
     endm
;                                       FUNCTION REQUEST 3AH
REM_DIR  macro  path
     mov    dx,offset path
     mov    ah,3AH
     call   loc_50H
     endm
;                                       FUNCTION REQUEST 3BH
CHANGE_DIR  macro  path
     mov    dx,offset path
     mov    ah,3BH
     call   loc_50H
     endm
;                                      FUNCTION REQUEST 3CH
```

```
CREATE_HANDLE  macro  path,attrib
     mov    dx,offset path
     mov    cx,attrib
     mov    ah,3CH
     call   loc_50H
     endm
;                                        FUNCTION REQUEST 3DH
OPEN_HANDLE  macro  path,access
     mov    dx,offset path
     mov    al,access
     mov     ah,3DH
     call   loc_50H
     endm
;                                        FUNCTION REQUEST 3EH
CLOSE_HANDLE  macro  handle
     mov    bx,handle
     mov    ah,3EH
     call   loc_50H
     endm
;                                        FUNCTION REQUEST 3FH
READ_HANDLE  macro  handle,buffer,bytes
     mov    bx,cs:handle
     mov    dx,offset buffer
     mov    cx,cs:bytes
     mov    ah,3FH
     call   cs:loc_50H
     endm
;                                       FUNCTION REQUEST 41H
DELETE_ENTRY  macro  path
     mov    dx,offset path
     mov    ah,41H
     call   loc_50H
     endm
;                                        FUNCTION REQUEST 42H
MOVE_PTR  macro  handle,high,low,method
     mov    bx,handle
     mov    cx,high
     mov    dx,low
     mov    al,method
     mov    ah,42H
     call   loc_50H
     endm
;                                        FUNCTION REQUEST 43H
CHANGE_MODE  macro  path,action,attrib
     mov    dx,offset path
     mov    al,action
     mov    cx,attrib
     mov    ah,43H
     call   loc_50H
     endm
;                                        FUNCTION REQUEST 44H
IO_CONTROL  macro  code,handle,buffer,attrib
     mov    al,code
     mov    bx,handle
     mov    dx,offset buffer
```

```
     mov    cx,attrib
     mov    ah,44H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 45H
XDUP  macro  handle
     mov    bx,handle
     mov    ah,45H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 46H
XDUP2  macro  handle1,handle2
     mov    bx,handle1
     mov    cx,handle2
     mov    ah,46H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 47H
GET_DIR  macro  drive,buffer
     mov    dl,drive
     mov    si,offset buffer
     mov    ah,47H
     call   loc_50H
     endm
;                                   FUNCTION REQUEST 48H
ALLOCATE_MEMORY  macro  paragraphs
     mov    bx,paragraphs
     mov    cl,4
     shr         bx,cl
     inc    bx
     mov    ah,48H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 49H
FREE_MEMORY  macro  seg_addr
     mov    ax,seg_addr
     mov    es,ax
     mov    ah,49H
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 4AH
SET_BLOCK  macro  last_byte
     mov    bx,offset last_byte
     mov    cl,4
     shr    bx,cl
     inc    bx
     mov    ah,4AH
     call   loc_50H
     endm
;                                    FUNCTION REQUEST 4BH
EXEC macro  path,command,parms,function
     mov    dx,offset path
     mov    bx,offset parms
     mov    word ptr parms[02h],offset command
     mov    word ptr parms[04h],cs
```

```
        mov     word ptr parms[06h],5ch
        mov     word ptr parms[08h],es
        mov     word ptr parms[0ah],6ch
        mov     word ptr parms[0ch],es
        mov     al,function
        mov     ah,4BH
        call    loc_50H
        endm

;                                       FUNCTION REQUEST 4CH
END_PROCESS  macro  return_code
        mov     al,return_code
        mov     ah,4CH
        call    loc_50H
        endm
;                                       FUNCTION REQUEST 4DH
WAIT  macro
        mov     ah,4DH
        call    loc_50H
        endm

;                                       FUNCTION REQUEST 4EH
FIND_FIRST_FILE  macro  path,attrib
        mov     dx,offset path
        mov     cx,attrib
        mov     ah,4EH
        call    loc_50H
        endm
;                                       FUNCTION REQUEST 4FH
FIND_NEXT_FILE  macro
        mov     ah,4FH
        call    loc_50H
        endm
;                                       FUNCTION REQUEST 54H
GET_VERIFY  macro
        mov     ah,54H
        call    loc_50H
        endm
;                                       FUNCTION REQUEST 56H
RENAME_FILE  macro  old_path,new_path
        mov     dx,offset old_path
        push    ds
        pop     es
        mov     di,offset new_path
        mov     ah,56H
        call    loc_50H
        endm
;                                       FUNCTION REQUEST 57H
GET_SET_DATE_TIME  macro  handle,action,time,date
        mov      bx,handle
        mov     al,action
        mov     cx,word ptr time
        mov     dx,word ptr date
        mov     ah,57H
        call    loc_50H
```

```
     endm
;
;
;********************
; General
;********************
;
DISPLAY_ASCIZ  macro  asciz_string
     local  search,found_it
     mov    bx,offset asciz_string

search:
     cmp    byte ptr [bx],0
     je     found_it
     inc    bx
     jmp short search

found_it:
     mov    byte ptr [bx],"$"
     display_asciz_string
     mov    byte ptr [bx],0
     display_char 0DH
     display_char 0AH
     endm
;
MOVE_STRING  macro  source,destination,count
     push   es
     push   ds
     pop    es
     assume es:code
     mov    si,offset source
     mov    di,offset destination
     mov    cx,count
 rep movs   es:destination,source
     assume es:nothing
     pop    es
     endm
;
CONVERT  macro  value,base,destination
     local  table,start
     jmp    start
table  db   "0123456789ABCDEF"

start:
     push   ax
     push   bx
     push   dx
     mov    al,value
     xor    ah,ah
     xor    bx,bx
     div    base
     mov    bl,al
     mov    al,cs:table[bx]
     mov    destination,al
     mov    bl,ah
```

```
     mov    destination[1],al
     pop    dx
     pop    bx
     pop    ax
     endm
;
CONVERT_TO_BINARY  macro  string,number,value
     local  ten,start,calc,mult,no_mult
     jmp    start
ten  db     10

start:
     mov    value,0
     xor    cx,cx
     mov    cl,number
     xor    si,si

calc:
     xor    ax,ax
     mov    al,string[si]
     sub    al,48
     cmp    cx,2
     jl     no_mult
     push   cx
     dec    cx

mult:
     mul    cs:ten
     loop   mult
     pop    cx

no_mult:
     add    value,ax
     inc    si
     loop   calc
     endm
;
CONVERT_DATE  macro  dir_entry
     mov    dx,word ptr dir_entry[24]
     mov    cl,5
     shr    dl,cl
     mov    dh,dir_entry[24]
     and    dh,1FH
     xor    cx,cx
     mov    cl,dir_entry[25]
     shr    cl,1
     add    cx,1980
     endm
;
PACK_DATE  macro  date
     local set_bit
;
; On entry: DH=day, DL=month, CX=(year-1980)
;
     sub   cx,1980
```

```
      push  cx
      mov   date,dh
      mov   cl,5
      shl   dl,cl
      pop   cx
      jnc   set_bit
      or    cl,80h

set_bit:
      or    date,dl
      rol   cl,1
      mov   date[1],cl
      endm
```

Chapter 2 MS-DOS 2.0 Device Drivers

2.1 What is a Device Driver? 2-1
2.2 Device Headers 2-3
2.2.1 Pointer to Next Device Field 2-3
2.2.2 Attribute Field 2-4
2.2.3 Strategy and Interrupt Routines 2-5
2.2.4 Name Field 2-5
2.3 How to Create a Device Driver 2-5
2.4 Installation of Device Drivers 2-6
2.5 Request Header 2-6
2.5.1 Unit Code 2-7
2.5.2 Command Code Field 2-7
2.5.3 MEDIA CHECK and BUILD BPB 2-8
2.5.4 Status Word 2-9
2.6 Function Call Parameters 2-11
2.6.1 INIT 2-12
2.6.2 MEDIA CHECK 2-12
2.6.3 BUILD BPB 2-13
2.6.4 Media Descriptor Byte 2-15
2.6.5 READ OR WRITE 2-16
2.6.6 NON DESTRUCTIVE READ NO WAIT 2-17
2.6.7 STATUS 2-18
2.6.8 FLUSH 2-18
2.7 The CLOCK Device 2-19
2.8 Example of Device Drivers 2-20
2.8.1 Block Device Driver 2-20
2.8.2 Character Device Driver 2-34

CHAPTER 2

MS-DOS 2.0 DEVICE DRIVERS

2.1 WHAT IS A DEVICE DRIVER?

A device driver is a binary file with all of the code in it to manipulate the hardware and provide a consistent interface to MS-DOS. In addition, it has a special header at the beginning that identifies it as a device, defines the strategy and interrupt entry points, and describes various attributes of the device.

NOTE

For device drivers, the file must not use the ORG 100H (like .COM files). Because it does not use the Program Segment Prefix, the device driver is simply loaded; therefore, the file must have an origin of zero (ORG 0 or no ORG statement).

There are two kinds of device drivers.

1. Character device drivers

2. Block device drivers

Character devices are designed to perform serial character I/O like CON, AUX, and PRN. These devices are named (i.e., CON, AUX, CLOCK, etc.), and users may open channels (handles or FCBs) to do I/O to them.

Block devices are the "disk drives" on the system. They can perform random I/O in pieces called blocks (usually the physical sector size). These devices are not named as the character devices are, and therefore cannot be opened directly. Instead they are identified via the drive letters (A, B, C, etc.).

Block devices also have units. A single driver may be responsible for one or more disk drives. For example, block device driver ALPHA may be responsible for drives A:,B:,C: and D:. This means that it has four units (0-3) defined and, therefore, takes up four drive letters. The position of the driver in the list of all drivers determines which units correspond to which driver letters. If driver ALPHA is the first block driver in the device list, and it defines 4 units (0-3), then they will be A:,B:,C: and D:. If BETA is the second block driver and defines three units (0-2), then they will be E:,F: and G:, and so on. MS-DOS 2.0 is not limited to 16 block device units, as were previous versions. The theoretical limit is 63 (26 - 1), but it should be noted that after 26 the drive letters are unconventional (such as], \, and ^).

NOTE

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

2.2 DEVICE HEADERS

A device header is required at the beginning of a device driver. A device header looks like this:

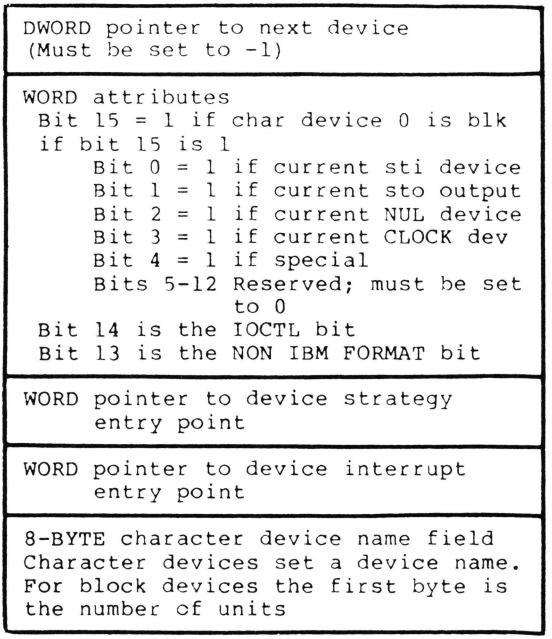


Figure 2.1. Sample Device Header

Note that the device entry points are words. They must be offsets from the same segment number used to point to this table. For example, if XXX:YYY points to the start of this table, then XXX:strategy and XXX:interrupt are the entry points.

2.2.1 Pointer To Next Device Field

The pointer to the next device header field is a double word field (offset followed by segment) that is set by MS-DOS to point at the next driver in the system list at the time the device driver is loaded. It is important that this field be set to -1 prior to load (when it is on the disk as a file) unless there is more than one device driver in the file. If there is more than one driver in the file, the first word of the double word pointer should be the offset of the next driver's Device Header.

NOTE

If there is more than one device driver in the .COM file, the last driver in the file must have the pointer to the next Device Header field set to -1.

2.2.2 Attribute Field

The attribute field is used to tell the system whether this device is a block or character device (bit 15). Most other bits are used to give selected character devices certain special treatment. (Note that these bits mean nothing on a block device). For example, assume that a user has a new device driver that he wants to be the standard input and output. Besides installing the driver, he must tell MS-DOS that he wants his new driver to override the current standard input and standard output (the CON device). This is accomplished by setting the attributes to the desired characteristics, so he would set bits 0 and 1 to 1 (note that they are separate!). Similarly, a new CLOCK device could be installed by setting that attribute. (Refer to Section 2.7, "The CLOCK Device," in this chapter for more information.) Although there is a NUL device attribute, the NUL device cannot be reassigned. This attribute exists so that MS-DOS can determine if the NUL device is being used.

The NON IBM FORMAT bit applies only to block devices and affects the operation of the BUILD BPB (BIOS Parameter Block) device call. (Refer to Section 2.5.3, "MEDIA CHECK And BUILD BPB," for further information on this call).

The other bit of interest is the IOCTL bit, which has meaning on character and block devices. This bit tells MS-DOS whether the device can handle control strings (via the IOCTL system call, Function 44H).

If a driver cannot process control strings, it should initially set this bit to 0. This tells MS-DOS to return an error if an attempt is made (via Function 44H) to send or receive control strings to this device. A device which can process control strings should initialize the IOCTL bit to 1. For drivers of this type, MS-DOS will make calls to the IOCTL INPUT and OUTPUT device functions to send and receive IOCTL strings.

The IOCTL functions allow data to be sent and received by the device for its own use (for example, to set baud rate, stop bits, and form length), instead of passing data over

the device channel as does a normal read or write. The interpretation of the passed information is up to the device, but it must not be treated as a normal I/O request.

2.2.3 Strategy And Interrupt Routines

These two fields are the pointers to the entry points of the strategy and interrupt routines. They are word values, so they must be in the same segment as the Device Header.

2.2.4 Name Field

This is an 8-byte field that contains the name of a character device or the number of units of a block device. If it is a block device, the number of units can be put in the first byte. This is optional, because MS-DOS will fill in this location with the value returned by the driver's INIT code. Refer to Section 2.4, "Installation of Device Drivers" in this chapter for more information.

2.3 HOW TO CREATE A DEVICE DRIVER

In order to create a device driver that MS-DOS can install, you must write a binary file with a Device Header at the beginning of the file. Note that for device drivers, the code should not be originated at 100H, but rather at 0. The link field (pointer to next Device Header) should be -1, unless there is more than one device driver in the file. The attribute field and entry points must be set correctly.

If it is a character device, the name field should be filled in with the name of that character device. The name can be any legal 8-character filename.

MS-DOS always processes installable device drivers before handling the default devices, so to install a new CON device, simply name the device CON. Remember to set the standard input device and standard output device bits in the attribute word on a new CON device. The scan of the device list stops on the first match, so the installable device driver takes precedence.

NOTE

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

2.4 INSTALLATION OF DEVICE DRIVERS

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

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

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

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

2.5 REQUEST HEADER

When MS-DOS calls a device driver to perform a function, it passes a Request Header in ES:BX to the strategy entry point. This is a fixed length header, followed by data pertinent to the operation being performed. Note that it is the device driver's responsibility to preserve the machine state (for example, save all registers on entry and restore them on exit). There is enough room on the stack when strategy or interrupt is called to do about 20 pushes. If more stack is needed, the driver should set up its own stack.

The following figure illustrates a Request Header.

REQUEST HEADER ->

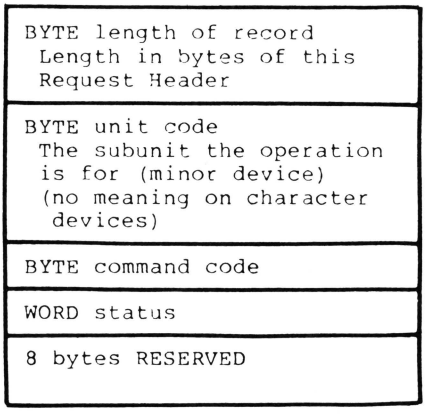


Figure 2.2. Request Header

2.5.1 Unit Code

The unit code field identifies which unit in your device driver the request is for. For example, if your device driver has 3 units defined, then the possible values of the unit code field would be 0, 1, and 2.

2.5.2 Command Code Field

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

Command Code	Function
0	INIT
1	MEDIA CHECK (Block only, NOP for character)
2	BUILD BPB " " " " "
3	IOCTL INPUT (Only called if device has IOCTL)
4	INPUT (Read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Char devs only)
6	INPUT STATUS " " "
7	INPUT FLUSH " " "
8	OUTPUT (Write)
9	OUTPUT (Write) with verify
10	OUTPUT STATUS " " "
11	OUTPUT FLUSH " " "
12	IOCTL OUTPUT (Only called if device has IOCTL)

2.5.3 MEDIA CHECK And BUILD BPB

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

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

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

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

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

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

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

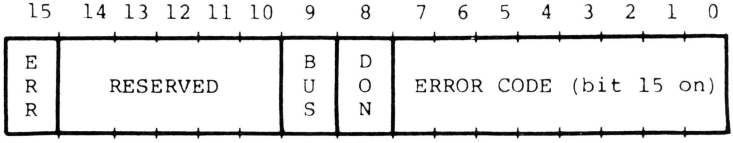
If Media Changed is returned

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

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

2.5.4 Status Word

The following figure illustrates the status word in the Request Header.



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

Bit 8 is the done bit. When set, it means the operation is complete. For MS-DOS 2.0, the driver sets it to 1 when it exits.

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

0 Write protect violation
1 Unknown Unit
2 Drive not ready
3 Unknown command
4 CRC error
5 Bad drive request structure length
6 Seek error
7 Unknown media
8 Sector not found
9 Printer out of paper
A Write fault
B Read Fault
C General failure

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

> For output on character devices: If bit 9 is 1 on return, a write request (if made) would wait for completion of a current request. If it is 0, there is no current request, and a write request (if made) would start immediately.

For input on character devices with a buffer: If bit 9 is 1 on return, a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the device buffer and a read would return quickly. It also indicates that something has been typed. MS-DOS assumes all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy=0 so that MS-DOS will not continuously wait for something to get into a buffer that does not exist.

One of the functions defined for each device is INIT. This routine is called only once when the device is installed. The INIT routine returns the BREAK ADDRESS, which is a DWORD pointer to the first free byte in memory. This pointer method can be used to delete initialization code that is only needed once, saving on space.

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

The number of units is returned. This determines logical device names. If the current maximum logical device letter is F at the time of the install call, and the INIT routine returns 4 as the number of units, then they will have logical names G, H, I and J. This mapping is determined by the position of the driver in the device list, and by the number of units on the device (stored in the first byte of the device name field).

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

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

Block devices may take several approaches; they may be dumb or smart. A dumb device defines a unit (and therefore an internal DOS structure) for each possible media drive combination. For example, unit 0 = drive 0 single side, unit 1 = drive 0 double side. For this approach, media descriptor bytes do not mean anything. A smart device allows multiple media per unit. In this case, the BPB table returned at INIT must define space large enough to accommodate the largest possible media supported. Smart drivers will use the media descriptor byte to pass information about what media is currently in a unit.

2.6 FUNCTION CALL PARAMETERS

All strategy routines are called with ES:BX pointing to the Request Header. The interrupt routines get the pointers to the Request Header from the queue that the strategy routines store them in. The command code in the Request Header tells the driver which function to perform.

NOTE

All DWORD pointers are stored offset first, then segment.

2.6.1 INIT

Command code = 0

INIT - ES:BX ->

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

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

NOTE

If there are multiple device drivers in a single .COM file, the ending address returned by the last INIT called will be the one MS-DOS uses. It is recommended that all of the device drivers in a single .COM file return the same ending address.

2.6.2 MEDIA CHECK

Command Code = 1

MEDIA CHECK - ES:BX ->

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

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

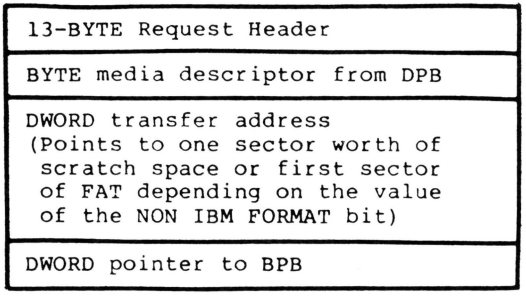
```
-1 Media has been changed
 0 Don't know if media has been changed
 1 Media has not been changed
```

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

2.6.3 BUILD BPB (BIOS Parameter Block)

Command code = 2

BUILD BPB - ES:BX ->

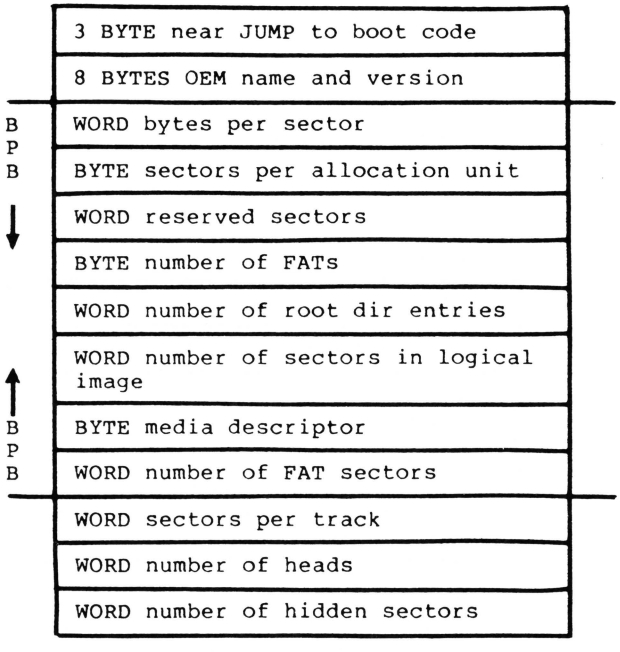


If the NON IBM FORMAT bit of the device is set, then the DWORD transfer address points to a one sector buffer, which can be used for any purpose. If the NON IBM FORMAT bit is 0, then this buffer contains the first sector of the first FAT and the driver must not alter this buffer.

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

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

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



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

2.6.4 Media Descriptor Byte

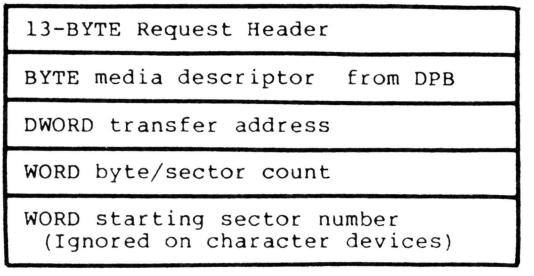
In MS-DOS 2.0, the media descriptor byte identifies different media. The media descriptor byte can be any value between 0 and FFH. It does not have to be the same as the FAT ID byte. The FAT ID byte, which is the first byte of the FAT, was used in MS-DOS 1.0 to determine the disk media and may be used as well under 2.0 disk device drivers. However, FAT ID bytes only have significance for IBM format-compatible block device drivers.

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

2.6.5 READ Or WRITE

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

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



In addition to setting the status word, the driver must set the sector count to the actual number of sectors (or bytes) transferred. No error check is performed on an IOCTL I/O call. The driver must always correctly set the return sector (byte) count to the actual number of bytes transferred.

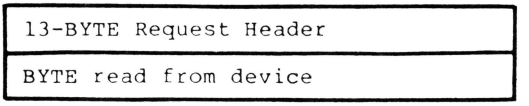
THE FOLLOWING APPLIES TO BLOCK DEVICE DRIVERS:

Under certain circumstances the BIOS may be asked to perform a write operation of 64K bytes, which seems to be a "wrap around" of the transfer address in the BIOS I/O packet. This request arises due to an optimization added to the write code in MS-DOS. It will only manifest on user writes that are within a sector size of 64K bytes on files "growing" past the current EOF. It is allowable for the BIOS to ignore the balance of the write that "wraps around" if it so chooses. For example, a write of 10000H bytes worth of sectors with a transfer address of XXX:1 could ignore the last two bytes. A user program can never request an I/O of more than FFFFH bytes and cannot wrap around (even to 0) in the transfer segment. Therefore, in this case, the last two bytes can be ignored.

2.6.6 NON DESTRUCTIVE READ NO WAIT

Command code = 5

NON DESRUCTIVE READ NO WAIT - ES:BX ->

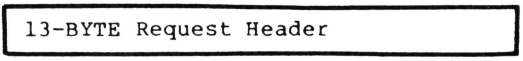


If the character device returns busy bit = 0 (characters in buffer), then the next character that would be read is returned. This character is not removed from the input buffer (hence the term "Non Destructive Read"). Basically, this call allows MS-DOS to look ahead one input character.

2.6.7 STATUS

Command codes = 6 and 10

STATUS Calls - ES:BX ->



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

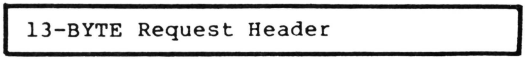
For output on character devices: If bit 9 is 1 on return, a write request (if made) would wait for completion of a current request. If it is 0, there is no current request and a write request (if made) would start immediately.

For input on character devices with a buffer: A return of 1 means, a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the devices buffer and a read would return quickly. A return of 0 also indicates that the user has typed something. MS-DOS assumes that all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy = 0 so that the DOS will not hang waiting for something to get into a buffer which doesn't exist.

2.6.8 FLUSH

Command codes = 7 and 11

FLUSH Calls - ES:BX ->



The FLUSH call tells the driver to flush (terminate) all pending requests. This call is used to flush the input queue on character devices.

2.7 THE CLOCK DEVICE

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

2.8 EXAMPLE OF DEVICE DRIVERS

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

2.8.1 Block Device Driver

```
;********************* A BLOCK DEVICE ******************

        TITLE   5 1/4" DISK DRIVER FOR SCP DISK-MASTER

;This driver is intended to drive up to four 5 1/4" drives
;hooked to the Seattle Computer Products DISK MASTER disk
;controller. All standard IBM PC formats are supported.


FALSE   EQU     0
TRUE    EQU     NOT FALSE

;The I/O port address of the DISK MASTER
DISK    EQU     0E0H
;DISK+0
;       1793    Command/Status
;DISK+1
;       1793    Track
;DISK+2
;       1793    Sector
;DISK+3
;       1793    Data
;DISK+4
;       Aux Command/Status
;DISK+5
;       Wait Sync

;Back side select bit
BACKBIT EQU     04H
;5 1/4" select bit
SMALBIT EQU     10H
;Double Density bit
DDBIT   EQU     08H

;Done bit in status register
DONEBIT EQU     01H

;Use table below to select head step speed.
;Step times for 5" drives
;are double that shown in the table.
;
;Step value    1771    1793
;
;    0          6ms     3ms
;    1          6ms     6ms
```

```
;    2          10ms     10ms
;    3          20ms     15ms
;
STPSPD  EQU     1

NUMERR  EQU     ERROUT-ERRIN

CR      EQU     0DH
LF      EQU     0AH

CODE    SEGMENT
ASSUME  CS:CODE,DS:NOTHING,ES:NOTHING,SS:NOTHING
;--------------------------------------------------
;
;       DEVICE HEADER
;
DRVDEV  LABEL   WORD
        DW      -1,-1
        DW      0000      ;IBM format-compatible, Block
        DW      STRATEGY
        DW      DRV$IN
DRVMAX  DB      4

DRVTBL  LABEL   WORD
        DW      DRV$INIT
        DW      MEDIA$CHK
        DW      GET$BPB
        DW      CMDERR
        DW      DRV$READ
        DW      EXIT
        DW      EXIT
        DW      EXIT
        DW      DRV$WRIT
        DW      DRV$WRIT
        DW      EXIT
        DW      EXIT
        DW      EXIT

;-------------------------------------
;
;       STRATEGY

PTRSAV  DD      0

STRATP  PROC    FAR
STRATEGY:
        MOV     WORD PTR [PTRSAV],BX
        MOV     WORD PTR [PTRSAV+2],ES
        RET
STRATP  ENDP

;-------------------------------------
;
;       MAIN ENTRY
```

```
CMDLEN  =       0       ;LENGTH OF THIS COMMAND
UNIT    =       1       ;SUB UNIT SPECIFIER
CMDC    =       2       ;COMMAND CODE
STATUS  =       3       ;STATUS
MEDIA   =       13      ;MEDIA DESCRIPTOR
TRANS   =       14      ;TRANSFER ADDRESS
COUNT   =       18      ;COUNT OF BLOCKS OR CHARACTERS
START   =       20      ;FIRST BLOCK TO TRANSFER

DRV$IN:
        PUSH    SI
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        PUSH    BX

        LDS     BX,[PTRSAV]     ;GET POINTER TO I/O PACKET

        MOV     AL,BYTE PTR [BX].UNIT   ;AL = UNIT CODE
        MOV     AH,BYTE PTR [BX].MEDIA  ;AH = MEDIA DESCRIP
        MOV     CX,WORD PTR [BX].COUNT  ;CX = COUNT
        MOV     DX,WORD PTR [BX].START  ;DX = START SECTOR
        PUSH    AX
        MOV     AL,BYTE PTR [BX].CMDC   ;Command code
        CMP     AL,11
        JA      CMDERRP                 ;Bad command
        CBW
        SHL     AX,1                    ;2 times command =
                                        ;word table index
        MOV     SI,OFFSET DRVTBL
        ADD     SI,AX                   ;Index into table
        POP     AX                      ;Get back media
                                        ;and unit

        LES     DI,DWORD PTR [BX].TRANS ;ES:DI = TRANSFER
                                        ;ADDRESS

        PUSH    CS
        POP     DS

ASSUME  DS:CODE

        JMP     WORD PTR [SI]             ;GO DO COMMAND

;---------------------------------------------------------
;
;       EXIT - ALL ROUTINES RETURN THROUGH THIS PATH
;
ASSUME  DS:NOTHING
CMDERRP:
```

```
        POP     AX                      ;Clean stack
CMDERR:
        MOV     AL,3                    ;UNKNOWN COMMAND ERROR
        JMP     SHORT ERR$EXIT

ERR$CNT:LDS     BX,[PTRSAV]
        SUB     WORD PTR [BX].COUNT,CX ;# OF SUCCESS. I/Os

ERR$EXIT:
;AL has error code
        MOV     AH,10000001B            ;MARK ERROR RETURN
        JMP     SHORT ERR1

EXITP   PROC    FAR

EXIT:   MOV     AH,00000001B
ERR1:   LDS     BX,[PTRSAV]
        MOV     WORD PTR [BX].STATUS,AX
                                        ;MARK OPERATION COMPLETE

        POP     BX
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     DX
        POP     CX
        POP     AX
        POP     SI
        RET                             ;RESTORE REGS AND RETURN
EXITP   ENDP

CURDRV  DB      -1

TRKTAB  DB      -1,-1,-1,-1

SECCNT  DW      0

DRVLIM  =       8       ;Number of sectors on device
SECLIM  =       13      ;MAXIMUM SECTOR
HDLIM   =       15      ;MAXIMUM HEAD

;WARNING - preserve order of drive and curhd!

DRIVE   DB      0       ;PHYSICAL DRIVE CODE
CURHD   DB      0       ;CURRENT HEAD
CURSEC  DB      0       ;CURRENT SECTOR
CURTRK  DW      0       ;CURRENT TRACK


;
MEDIA$CHK:              ;Always indicates Don't know
ASSUME  DS:CODE
        TEST    AH,00000100B        ;TEST IF MEDIA REMOVABLE
        JZ      MEDIA$EXT
```

```
        XOR     DI,DI                   ;SAY I DON'T KNOW
MEDIA$EXT:
        LDS     BX,[PTRSAV]
        MOV     WORD PTR [BX].TRANS,DI
        JMP     EXIT

BUILD$BPB:
ASSUME  DS:CODE
        MOV     AH,BYTE PTR ES:[DI]     ;GET FAT ID BYTE
        CALL    BUILDBP                 ;TRANSLATE
SETBPB: LDS     BX,[PTRSAV]
        MOV     [BX].MEDIA,AH
        MOV     [BX].COUNT,DI
        MOV     [BX].COUNT+2,CS
        JMP     EXIT

BUILDBP:
ASSUME  DS:NOTHING
;AH is media byte on entry
;DI points to correct BPB on return
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    BX
        MOV     CL,AH           ;SAVE MEDIA
        AND     CL,0F8H         ;NORMALIZE
        CMP     CL,0F8H         ;COMPARE WITH GOOD MEDIA BYTE
        JZ      GOODID
        MOV     AH,0FEH         ;DEFAULT TO 8-SECTOR,
                                ;SINGLE-SIDED
GOODID:
        MOV     AL,1            ;SET NUMBER OF FAT SECTORS
        MOV     BX,64*256+8     ;SET DIR ENTRIES AND SECTOR MAX
        MOV     CX,40*8         ;SET SIZE OF DRIVE
        MOV     DX,01*256+1     ;SET HEAD LIMIT & SEC/ALL UNIT
        MOV     DI,OFFSET DRVBPB
        TEST    AH,00000010B    ;TEST FOR 8 OR 9 SECTOR
        JNZ     HAS8            ;NZ = HAS 8 SECTORS
        INC     AL              ;INC NUMBER OF FAT SECTORS
        INC     BL              ;INC SECTOR MAX
        ADD     CX,40           ;INCREASE SIZE
HAS8:   TEST    AH,00000001B    ;TEST FOR 1 OR 2 HEADS
        JZ      HAS1            ;Z = 1 HEAD
        ADD     CX,CX           ;DOUBLE SIZE OF DISK
        MOV     BH,112          ;INCREASE # OF DIREC. ENTRIES
        INC     DH              ;INC SEC/ALL UNIT
        INC     DL              ;INC HEAD LIMIT
HAS1:   MOV     BYTE PTR [DI].2,DH
        MOV     BYTE PTR [DI].6,BH
        MOV     WORD PTR [DI].8,CX
        MOV     BYTE PTR [DI].10,AH
        MOV     BYTE PTR [DI].11,AL
        MOV     BYTE PTR [DI].13,BL
        MOV     BYTE PTR [DI].15,DL
        POP     BX
```

```
        POP     DX
        POP     CX
        POP     AX
        RET


;--------------------------------------------------------
;
;       DISK I/O HANDLERS
;
;ENTRY:
;       AL = DRIVE NUMBER (0-3)
;       AH = MEDIA DESCRIPTOR
;       CX = SECTOR COUNT
;       DX = FIRST SECTOR
;       DS = CS
;       ES:DI = TRANSFER ADDRESS
;EXIT:
;       IF SUCCESSFUL CARRY FLAG = 0
;         ELSE CF=1 AND AL CONTAINS (MS-DOS) ERROR CODE,
;         CX # sectors NOT transferred

DRV$READ:
ASSUME  DS:CODE
        JCXZ    DSKOK
        CALL    SETUP
        JC      DSK$IO
        CALL    DISKRD
        JMP     SHORT DSK$IO

DRV$WRIT:
ASSUME  DS:CODE
        JCXZ    DSKOK
        CALL    SETUP
        JC      DSK$IO
        CALL    DISKWRT
ASSUME  DS:NOTHING
DSK$IO: JNC     DSKOK
        JMP     ERR$CNT
DSKOK:  JMP     EXIT


SETUP:
ASSUME  DS:CODE
;Input same as above
;On output
; ES:DI = Trans addr
; DS:BX Points to BPB
; Carry set if error (AL is error code (MS-DOS))
; else
;       [DRIVE] = Drive number (0-3)
;       [SECCNT] = Sectors to transfer
;       [CURSEC] = Sector number of start of I/O
;       [CURHD]  = Head number of start of I/O   ;Set
;       [CURTRK] = Track # of start of I/O ;Seek performed
```

```
; All other registers destroyed

        XCHG    BX,DI                   ;ES:BX = TRANSFER ADDRESS
        CALL    BUILDBP                 ;DS:DI = PTR TO B.P.B
        MOV     SI,CX
        ADD     SI,DX
        CMP     SI,WORD PTR [DI].DRVLIM
                                        ;COMPARE AGAINST DRIVE MAX
        JBE     INRANGE
        MOV     AL,8
        STC
        RET

INRANGE:
        MOV     [DRIVE],AL
        MOV     [SECCNT],CX     ;SAVE SECTOR COUNT
        XCHG    AX,DX           ;SET UP LOGICAL SECTOR
                                ;FOR DIVIDE
        XOR     DX,DX
        DIV     WORD PTR [DI].SECLIM ;DIVIDE BY SEC PER TRACK
        INC     DL
        MOV     [CURSEC],DL             ;SAVE CURRENT SECTOR
        MOV     CX,WORD PTR [DI].HDLIM ;GET NUMBER OF HEADS
        XOR     DX,DX   ;DIVIDE TRACKS BY HEADS PER CYLINDER
        DIV     CX
        MOV     [CURHD],DL      ;SAVE CURRENT HEAD
        MOV     [CURTRK],AX     ;SAVE CURRENT TRACK
SEEK:
        PUSH    BX              ;Xaddr
        PUSH    DI              ;BPB pointer
        CALL    CHKNEW          ;Unload head if change drives
        CALL    DRIVESEL
        MOV     BL,[DRIVE]
        XOR     BH,BH           ;BX drive index
        ADD     BX,OFFSET TRKTAB        ;Get current track
        MOV     AX,[CURTRK]
        MOV     DL,AL           ;Save desired track
        XCHG    AL,DS:[BX]      ;Make desired track current
        OUT     DISK+1,AL       ;Tell Controller current track
        CMP     AL,DL           ;At correct track?
        JZ      SEEKRET         ;Done if yes
        MOV     BH,2            ;Seek retry count
        CMP     AL,-1           ;Position Known?
        JNZ     NOHOME          ;If not home head
TRYSK:
        CALL    HOME
        JC      SEEKERR
NOHOME:
        MOV     AL,DL
        OUT     DISK+3,AL       ;Desired track
        MOV     AL,1CH+STPSPD   ;Seek
        CALL    DCOM
        AND     AL,98H    ;Accept not rdy, seek, & CRC errors
        JZ      SEEKRET
        JS      SEEKERR         ;No retries if not ready
```

```
        DEC     BH
        JNZ     TRYSK
SEEKERR:
        MOV     BL,[DRIVE]
        XOR     BH,BH           ;BX drive index
        ADD     BX,OFFSET TRKTAB        ;Get current track
        MOV     BYTE PTR DS:[BX],-1     ;Make current track
                                        ;unknown
        CALL    GETERRCD
        MOV     CX,[SECCNT]     ;Nothing transferred
        POP     BX              ;BPB pointer
        POP     DI              ;Xaddr
        RET

SEEKRET:
        POP     BX              ;BPB pointer
        POP     DI              ;Xaddr
        CLC
        RET

;-----------------------------------------------
;
;       READ
;

DISKRD:
ASSUME  DS:CODE
        MOV     CX,[SECCNT]
RDLP:
        CALL    PRESET
        PUSH    BX
        MOV     BL,10               ;Retry count
        MOV     DX,DISK+3           ;Data port
RDAGN:
        MOV     AL,80H              ;Read command
        CLI                         ;Disable for 1793
        OUT     DISK,AL             ;Output read command
        MOV     BP,DI               ;Save address for retry
        JMP     SHORT RLOOPENTRY
RLOOP:
        STOSB
RLOOPENTRY:
        IN      AL,DISK+5           ;Wait for DRQ or INTRQ
        SHR     AL,1
        IN      AL,DX               ;Read data
        JNC     RLOOP
        STI                         ;Ints OK now
        CALL    GETSTAT
        AND     AL,9CH
        JZ      RDPOP               ;Ok
        MOV     DI,BP               ;Get back transfer
        DEC     BL
        JNZ     RDAGN
        CMP     AL,10H              ;Record not found?
        JNZ     GOT_CODE            ;No
```

```
        MOV     AL,1                ;Map it
GOT_CODE:
        CALL    GETERRCD
        POP     BX
        RET

RDPOP:
        POP     BX
        LOOP    RDLP
        CLC
        RET


;-----------------------------------------------
;
;       WRITE
;

DISKWRT:
ASSUME  DS:CODE
        MOV     CX,[SECCNT]
        MOV     SI,DI
        PUSH    ES
        POP     DS
ASSUME  DS:NOTHING
WRLP:
        CALL    PRESET
        PUSH    BX
        MOV     BL,10                   ;Retry count
        MOV     DX,DISK+3               ;Data port
WRAGN:
        MOV     AL,0A0H             ;Write command
        CLI                         ;Disable for 1793
        OUT     DISK,AL             ;Output write command
        MOV     BP,SI               ;Save address for retry
WRLOOP:
        IN      AL,DISK+5
        SHR     AL,1
        LODSB                       ;Get data
        OUT     DX,AL               ;Write data
        JNC     WRLOOP
        STI                         ;Ints OK now
        DEC     SI
        CALL    GETSTAT
        AND     AL,0FCH
        JZ      WRPOP               ;Ok
        MOV     SI,BP               ;Get back transfer
        DEC     BL
        JNZ     WRAGN
        CALL    GETERRCD
        POP     BX
        RET

WRPOP:
        POP     BX
```

```
        LOOP    WRLP
        CLC
        RET


PRESET:
ASSUME  DS:NOTHING
        MOV     AL,[CURSEC]
        CMP     AL,CS:[BX].SECLIM
        JBE     GOTSEC
        MOV     DH,[CURHD]
        INC     DH
        CMP     DH,CS:[BX].HDLIM
        JB      SETHEAD             ;Select new head
        CALL    STEP                ;Go on to next track
        XOR     DH,DH               ;Select head zero
SETHEAD:
        MOV     [CURHD],DH
        CALL    DRIVESEL
        MOV     AL,1                ;First sector
        MOV     [CURSEC],AL         ;Reset CURSEC
GOTSEC:
        OUT     DISK+2,AL       ;Tell controller which sector
        INC     [CURSEC]        ;We go on to next sector
        RET

STEP:
ASSUME  DS:NOTHING
        MOV     AL,58H+STPSPD   ;Step in w/ update, no verify
        CALL    DCOM
        PUSH    BX
        MOV     BL,[DRIVE]
        XOR     BH,BH               ;BX drive index
        ADD     BX,OFFSET TRKTAB        ;Get current track
        INC     BYTE PTR CS:[BX]        ;Next track
        POP     BX
        RET

HOME:
ASSUME  DS:NOTHING
        MOV     BL,3
TRYHOM:
        MOV     AL,0CH+STPSPD   ;Restore with verify
        CALL    DCOM
        AND     AL,98H
        JZ      RET3
        JS      HOMERR          ;No retries if not ready
        PUSH    AX              ;Save real error code
        MOV     AL,58H+STPSPD   ;Step in w/ update no verify
        CALL    DCOM
        DEC     BL
        POP     AX              ;Get back real error code
        JNZ     TRYHOM
HOMERR:
        STC
```

```
RET3:   RET


CHKNEW:
ASSUME  DS:NOTHING
        MOV     AL,[DRIVE]      ;Get disk drive number
        MOV     AH,AL
        XCHG    AL,[CURDRV]     ;Make new drive current.
        CMP     AL,AH           ;Changing drives?
        JZ      RET1            ;No
; If changing drives, unload head so the head load delay
;one-shot will fire again. Do it by seeking to the same
;track with the H bit reset.
;
        IN      AL,DISK+1       ;Get current track number
        OUT     DISK+3,AL       ;Make it the track to seek
        MOV     AL,10H          ;Seek and unload head

DCOM:
ASSUME  DS:NOTHING
        OUT     DISK,AL
        PUSH    AX
        AAM                     ;Delay 10 microseconds
        POP     AX
GETSTAT:
        IN      AL,DISK+4
        TEST    AL,DONEBIT
        JZ      GETSTAT
        IN      AL,DISK
RET1:   RET

DRIVESEL:
ASSUME  DS:NOTHING
;Select the drive based on current info
;Only AL altered
        MOV     AL,[DRIVE]
        OR      AL,SMALBIT + DDBIT      ;5 1/4" IBM PC disks
        CMP     [CURHD],0
        JZ      GOTHEAD
        OR      AL,BACKBIT      ;Select side 1
GOTHEAD:
        OUT     DISK+4,AL       ;Select drive and side
        RET

GETERRCD:
ASSUME  DS:NOTHING
        PUSH    CX
        PUSH    ES
        PUSH    DI
        PUSH    CS
        POP     ES              ;Make ES the local segment
        MOV     CS:[LSTERR],AL ;Terminate list w/ error code
        MOV     CX,NUMERR       ;Number of error conditions
        MOV     DI,OFFSET ERRIN ;Point to error conditions
        REPNE   SCASB
```

```
        MOV     AL,NUMERR-1[DI] ;Get translation
        STC                     ;Flag error condition
        POP     DI
        POP     ES
        POP     CX
        RET                     ;and return

;*********************************************************
;       BPB FOR AN IBM FLOPPY DISK, VARIOUS PARAMETERS ARE
;       PATCHED BY BUILDBP TO REFLECT THE TYPE OF MEDIA
;       INSERTED
;       This is a nine sector single side BPB
DRVBPB:
        DW      512             ;Physical sector size in bytes
        DB      1               ;Sectors/allocation unit
        DW      1               ;Reserved sectors for DOS
        DB      2               ;# of allocation tables
        DW      64              ;Number directory entries
        DW      9*40            ;Number 512-byte sectors
        DB      11111100B       ;Media descriptor
        DW      2               ;Number of FAT sectors
        DW      9               ;Sector limit
        DW      1               ;Head limit


INITAB  DW      DRVBPB                  ;Up to four units
        DW      DRVBPB
        DW      DRVBPB
        DW      DRVBPB

ERRIN:  ;DISK ERRORS RETURNED FROM THE 1793 CONTROLER
        DB      80H             ;NO RESPONSE
        DB      40H             ;Write protect
        DB      20H             ;Write Fault
        DB      10H             ;SEEK error
        DB      8               ;CRC error
        DB      1               ;Mapped from 10H
                                ;(record not found) on READ
LSTERR  DB      0               ;ALL OTHER ERRORS

ERROUT: ;RETURNED ERROR CODES CORRESPONDING TO ABOVE
        DB      2               ;NO RESPONSE
        DB      0               ;WRITE ATTEMPT
                                ;ON WRITE-PROTECT DISK
        DB      0AH             ;WRITE FAULT
        DB      6               ;SEEK FAILURE
        DB      4               ;BAD CRC
        DB      8               ;SECTOR NOT FOUND
        DB      12              ;GENERAL ERROR


DRV$INIT:
;
; Determine number of physical drives by reading CONFIG.SYS
;
```

```
ASSUME  DS:CODE
        PUSH    DS
        LDS     SI,[PTRSAV]
ASSUME  DS:NOTHING
        LDS     SI,DWORD PTR [SI.COUNT] ;DS:SI points to
                                        ;CONFIG.SYS
SCAN_LOOP:
        CALL    SCAN_SWITCH
        MOV     AL,CL
        OR      AL,AL
        JZ      SCAN4
        CMP     AL,"s"
        JZ      SCAN4

WERROR: POP     DS
ASSUME  DS:CODE
        MOV     DX,OFFSET ERRMSG2
WERROR2: MOV    AH,9
        INT     21H
        XOR     AX,AX
        PUSH    AX                      ;No units
        JMP     SHORT ABORT

BADNDRV:
        POP     DS
        MOV     DX,OFFSET ERRMSG1
        JMP     WERROR2

SCAN4:
ASSUME  DS:NOTHING
;BX is number of floppies
        OR      BX,BX
        JZ      BADNDRV                 ;User error
        CMP     BX,4
        JA      BADNDRV                 ;User error
        POP     DS
ASSUME  DS:CODE
        PUSH    BX                      ;Save unit count
ABORT:  LDS     BX,[PTRSAV]
ASSUME  DS:NOTHING
        POP     AX
        MOV     BYTE PTR [BX].MEDIA,AL          ;Unit count
        MOV     [DRVMAX],AL
        MOV     WORD PTR [BX].TRANS,OFFSET DRV$INIT ;SET
                                                ;BREAK ADDRESS
        MOV     [BX].TRANS+2,CS
        MOV     WORD PTR [BX].COUNT,OFFSET INITAB
                                ;SET POINTER TO BPB ARRAY
        MOV     [BX].COUNT+2,CS
        JMP     EXIT
;
; PUT SWITCH IN CL, VALUE IN BX
;
SCAN_SWITCH:
        XOR     BX,BX
```

```
        MOV     CX,BX
        LODSB
        CMP     AL,10
        JZ      NUMRET
        CMP     AL,"-"
        JZ      GOT_SWITCH
        CMP     AL,"/"
        JNZ     SCAN_SWITCH
GOT_SWITCH:
        CMP     BYTE PTR [SI+1],":"
        JNZ     TERROR
        LODSB
        OR      AL,20H          ; CONVERT TO LOWER CASE
        MOV     CL,AL           ; GET SWITCH
        LODSB                   ; SKIP ":"
;
;  GET NUMBER POINTED TO BY [SI]
;
;  WIPES OUT AX,DX ONLY     BX RETURNS NUMBER
;
GETNUM1:LODSB
        SUB     AL,"0"
        JB      CHKRET
        CMP     AL,9
        JA      CHKRET
        CBW
        XCHG    AX,BX
        MOV     DX,10
        MUL     DX
        ADD     BX,AX
        JMP     GETNUM1

CHKRET: ADD     AL,"0"
        CMP     AL," "
        JBE     NUMRET
        CMP     AL,"-"
        JZ      NUMRET
        CMP     AL,"/"
        JZ      NUMRET
TERROR:
        POP     DS              ; GET RID OF RETURN ADDRESS
        JMP     WERROR
NUMRET: DEC     SI
        RET

ERRMSG1 DB      "SMLDRV: Bad number of drives",13,10,"$"
ERRMSG2 DB      "SMLDRV: Invalid parameter",13,10,"$"
CODE    ENDS
        END
```

2.8.2 Character Device Driver

The following program illustrates a character device driver program.

```
;******************** A CHARACTER DEVICE ********************

TITLE   VT52 CONSOLE FOR 2.0    (IBM)

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;       IBM ADDRESSES FOR I/O
;
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

        CR=13               ;CARRIAGE RETURN
        BACKSP=8            ;BACKSPACE
        ESC=1BH
        BRKADR=6CH          ;006C  BREAK VECTOR ADDRESS
        ASNMAX=200          ;SIZE OF KEY ASSIGNMENT BUFFER

CODE    SEGMENT BYTE

   ASSUME CS:CODE,DS:NOTHING,ES:NOTHING
;-------------------------------------------------------------
;
;       C O N - CONSOLE DEVICE DRIVER
;
CONDEV:                             ;HEADER FOR DEVICE "CON"
        DW      -1,-1
        DW      1000000000010011B   ;CON IN AND CON OUT
        DW      STRATEGY
        DW      ENTRY
        DB      'CON

;-------------------------------------------------------------
;
;       COMMAND JUMP TABLES
CONTBL:
        DW      CON$INIT
        DW      EXIT
        DW      EXIT
        DW      CMDERR
        DW      CON$READ
        DW      CON$RDND
        DW      EXIT
        DW      CON$FLSH
        DW      CON$WRIT
        DW      CON$WRIT
        DW      EXIT
        DW      EXIT

CMDTABL DB      'A'
```

```
        DW      CUU             ;cursor up
        DB      'B'
        DW      CUD             ;cursor down
        DB      'C'
        DW      CUF             ;cursor forward
        DB      'D'
        DW      CUB             ;cursor back
        DB      'H'
        DW      CUH             ;cursor position
        DB      'J'
        DW      ED              ;erase display
        DB      'K'
        DW      EL              ;erase line
        DB      'Y'
        DW      CUP             ;cursor position
        DB      'j'
        DW      PSCP            ;save cursor position
        DB      'k'
        DW      PRCP            ;restore cursor position
        DB      'y'
        DW      RM              ;reset mode
        DB      'x'
        DW      SM              ;set mode
        DB      00

PAGE
;-----------------------------------------------------
;
;       Device entry point
;
CMDLEN  =       0       ;LENGTH OF THIS COMMAND
UNIT    =       1       ;SUB UNIT SPECIFIER
CMD     =       2       ;COMMAND CODE
STATUS  =       3       ;STATUS
MEDIA   =       13      ;MEDIA DESCRIPTOR
TRANS   =       14      ;TRANSFER ADDRESS
COUNT   =       18      ;COUNT OF BLOCKS OR CHARACTERS
START   =       20      ;FIRST BLOCK TO TRANSFER

PTRSAV  DD      0

STRATP  PROC    FAR

STRATEGY:
        MOV     WORD PTR CS:[PTRSAV],BX
        MOV     WORD PTR CS:[PTRSAV+2],ES
        RET

STRATP  ENDP

ENTRY:
        PUSH    SI
        PUSH    AX
        PUSH    CX
        PUSH    DX
```

```
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        PUSH    BX

        LDS     BX,CS:[PTRSAV]  ;GET POINTER TO I/O PACKET

        MOV     CX,WORD PTR DS:[BX].COUNT     ;CX = COUNT

        MOV     AL,BYTE PTR DS:[BX].CMD
        CBW
        MOV     SI,OFFSET CONTBL
        ADD     SI,AX
        ADD     SI,AX
        CMP     AL,11
        JA      CMDERR

        LES     DI,DWORD PTR DS:[BX].TRANS

        PUSH    CS
        POP     DS

        ASSUME  DS:CODE

        JMP     WORD PTR [SI]                 ;GO DO COMMAND

PAGE
;=====================================================
;=
;=      SUBROUTINES SHARED BY MULTIPLE DEVICES
;=
;=====================================================
;-----------------------------------------------------
;
;       EXIT - ALL ROUTINES RETURN THROUGH THIS PATH
;
BUS$EXIT:                                   ;DEVICE BUSY EXIT
        MOV     AH,00000011B
        JMP     SHORT ERR1

CMDERR:
        MOV     AL,3                  ;UNKNOWN COMMAND ERROR

ERR$EXIT:
        MOV     AH,10000001B              ;MARK ERROR RETURN
        JMP     SHORT ERR1

EXITP   PROC    FAR

EXIT:   MOV     AH,00000001B
ERR1:   LDS     BX,CS:[PTRSAV]
        MOV     WORD PTR [BX].STATUS,AX ;MARK
                                        ;OPERATION COMPLETE
```

```
        POP     BX
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     DX
        POP     CX
        POP     AX
        POP     SI
        RET                             ;RESTORE REGS AND RETURN
EXITP   ENDP
;-----------------------------------------------
;
;       BREAK KEY HANDLING
;
BREAK:
        MOV     CS:ALTAH,3              ;INDICATE BREAK KEY SET
INTRET: IRET

PAGE
;
;       WARNING - Variables are very order dependent,
                  so be careful when adding new ones!
;
WRAP    DB      0                       ; 0 = WRAP, 1 = NO WRAP
STATE   DW      S1
MODE    DB      3
MAXCOL  DB      79
COL     DB      0
ROW     DB      0
SAVCR   DW      0
ALTAH   DB      0                       ;Special key handling

;--------------------------------------------------------
;
;    CHROUT - WRITE OUT CHAR IN AL USING CURRENT ATTRIBUTE
;
ATTRW   LABEL   WORD
ATTR    DB      00000111B               ;CHARACTER ATTRIBUTE
BPAGE   DB      0                       ;BASE PAGE
base    dw      0b800h

chrout: cmp     al,13
        jnz     trylf
        mov     [col],0
        jmp     short setit

trylf:  cmp     al,10
        jz      lf
        cmp     al,7
        jnz     tryback
torom:
        mov     bx,[attrw]
        and     bl,7
        mov     ah,14
```

```
        int     10h
ret5:   ret

tryback:
        cmp     al,8
        jnz     outchr
        cmp     [col],0
        jz      ret5
        dec     [col]
        jmp     short setit

outchr:
        mov     bx,[attrw]
        mov     cx,1
        mov     ah,9
        int     10h
        inc     [col]
        mov     al,[col]
        cmp     al,[maxcol]
        jbe     setit
        cmp     [wrap],0
        jz      outchr1
        dec     [col]
        ret
outchr1:
        mov     [col],0
lf:     inc     [row]
        cmp     [row],24
        jb      setit
        mov     [row],23
        call    scroll

setit:  mov     dh,row
        mov     dl,col
        xor     bh,bh
        mov     ah,2
        int     10h
        ret

scroll: call    getmod
        cmp     al,2
        jz      myscroll
        cmp     al,3
        jz      myscroll
        mov     al,10
        jmp     torom
myscroll:
        mov     bh,[attr]
        mov     bl,' '
        mov     bp,80
        mov     ax,[base]
        mov     es,ax
        mov     ds,ax
        xor     di,di
        mov     si,160
```

```
        mov     cx,23*80
        cld
        cmp     ax,0b800h
        jz      colorcard

        rep     movsw
        mov     ax,bx
        mov     cx,bp
        rep     stosw
sret:   push    cs
        pop     ds
        ret

colorcard:
        mov     dx,3dah
wait2:  in      al,dx
        test    al,8
        jz      wait2
        mov     al,25h
        mov     dx,3d8h
        out     dx,al           ;turn off video
        rep     movsw
        mov     ax,bx
        mov     cx,bp
        rep     stosw
        mov     al,29h
        mov     dx,3d8h
        out     dx,al           ;turn on video
        jmp     sret

GETMOD: MOV     AH,15
        INT     16              ;get column information
        MOV     BPAGE,BH
        DEC     AH
        MOV     WORD PTR MODE,AX
        RET
;--------------------------------------------------------
;
;       CONSOLE READ ROUTINE
;
CON$READ:
        JCXZ    CON$EXIT
CON$LOOP:
        PUSH    CX              ;SAVE COUNT
        CALL    CHRIN           ;GET CHAR IN AL
        POP     CX
        STOSB                   ;STORE CHAR AT ES:DI
        LOOP    CON$LOOP
CON$EXIT:
        JMP     EXIT
;--------------------------------------------------------
;
;       INPUT SINGLE CHAR INTO AL
;
CHRIN:  XOR     AX,AX
```

```
        XCHG    AL,ALTAH        ;GET CHARACTER & ZERO ALTAH
        OR      AL,AL
        JNZ     KEYRET

INAGN:  XOR     AH,AH
        INT     22
ALT10:
        OR      AX,AX           ;Check for non-key after BREAK
        JZ      INAGN
        OR      AL,AL           ;SPECIAL CASE?
        JNZ     KEYRET
        MOV     ALTAH,AH            ;STORE SPECIAL KEY
KEYRET: RET
;--------------------------------------------------------
;
;       KEYBOARD NON DESTRUCTIVE READ, NO WAIT
;
CON$RDND:
        MOV     AL,[ALTAH]
        OR      AL,AL
        JNZ     RDEXIT

RD1:    MOV     AH,1
        INT     22
        JZ      CONBUS
        OR      AX,AX
        JNZ     RDEXIT
        MOV     AH,0
        INT     22
        JMP     CON$RDND

RDEXIT: LDS     BX,[PTRSAV]
        MOV     [BX].MEDIA,AL
EXVEC:  JMP     EXIT
CONBUS: JMP     BUS$EXIT
;--------------------------------------------------------
;
;       KEYBOARD FLUSH ROUTINE
;
CON$FLSH:
        MOV     [ALTAH],0           ;Clear out holding buffer

        PUSH    DS
        XOR     BP,BP
        MOV     DS,BP                       ;Select segment 0
        MOV     DS:BYTE PTR 41AH,1EH        ;Reset KB queue head
                                            ;pointer
        MOV     DS:BYTE PTR 41CH,1EH        ;Reset tail pointer
        POP     DS
        JMP     EXVEC
;--------------------------------------------------------
;
;       CONSOLE WRITE ROUTINE
;
CON$WRIT:
```

```
        JCXZ    EXVEC
        PUSH    CX
        MOV     AH,3            ;SET CURRENT CURSOR POSITION
        XOR     BX,BX
        INT     16
        MOV     WORD PTR [COL],DX
        POP     CX

CON$LP: MOV     AL,ES:[DI]      ;GET CHAR
        INC     DI
        CALL    OUTC            ;OUTPUT CHAR
        LOOP    CON$LP          ;REPEAT UNTIL ALL THROUGH
        JMP     EXVEC

COUT:   STI
        PUSH    DS
        PUSH    CS
        POP     DS
        CALL    OUTC
        POP     DS
        IRET

OUTC:   PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    ES
        PUSH    BP
        CALL    VIDEO
        POP     BP
        POP     ES
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET


;--------------------------------------------------------
;
;       OUTPUT SINGLE CHAR IN AL TO VIDEO DEVICE
;
VIDEO:  MOV     SI,OFFSET STATE
        JMP     [SI]

S1:     CMP     AL,ESC                  ;ESCAPE SEQUENCE?
        JNZ     S1B
        MOV     WORD PTR [SI],OFFSET S2
        RET

S1B:    CALL    CHROUT
S1A:    MOV     WORD PTR [STATE],OFFSET S1
        RET
```

```
S2:     PUSH    AX
        CALL    GETMOD
        POP     AX
        MOV     BX,OFFSET CMDTABL-3
S7A:    ADD     BX,3
        CMP     BYTE PTR [BX],0
        JZ      S1A
        CMP     BYTE PTR [BX],AL
        JNZ     S7A
        JMP     WORD PTR [BX+1]

MOVCUR: CMP     BYTE PTR [BX],AH
        JZ      SETCUR
        ADD     BYTE PTR [BX],AL
SETCUR: MOV     DX,WORD PTR COL
        XOR     BX,BX
        MOV     AH,2
        INT     16
        JMP     S1A

CUP:    MOV     WORD PTR [SI],OFFSET CUP1
        RET
CUP1:   SUB     AL,32
        MOV     BYTE PTR [ROW],AL
        MOV     WORD PTR [SI],OFFSET CUP2
        RET
CUP2:   SUB     AL,32
        MOV     BYTE PTR [COL],AL
        JMP     SETCUR

SM:     MOV     WORD PTR [SI],OFFSET S1A
        RET

CUH:    MOV     WORD PTR COL,0
        JMP     SETCUR

CUF:    MOV     AH,MAXCOL
        MOV     AL,1
CUF1:   MOV     BX,OFFSET COL
        JMP     MOVCUR

CUB:    MOV     AX,00FFH
        JMP     CUF1

CUU:    MOV     AX,00FFH
CUU1:   MOV     BX,OFFSET ROW
        JMP     MOVCUR

CUD:    MOV     AX,23*256+1
        JMP     CUU1
```

```
PSCP:   MOV     AX,WORD PTR COL
        MOV     SAVCR,AX
        JMP     SETCUR

PRCP:   MOV     AX,SAVCR
        MOV     WORD PTR COL,AX
        JMP     SETCUR

ED:     CMP     BYTE PTR [ROW],24
        JAE     EL1

        MOV     CX,WORD PTR COL
        MOV     DH,24
        JMP     ERASE

EL1:    MOV     BYTE PTR [COL],0
EL:     MOV     CX,WORD PTR [COL]
EL2:    MOV     DH,CH
ERASE:  MOV     DL,MAXCOL
        MOV     BH,ATTR
        MOV     AX,0600H
        INT     16
ED3:    JMP     SETCUR


RM:     MOV     WORD PTR [SI],OFFSET RM1
        RET
RM1:    XOR     CX,CX
        MOV     CH,24
        JMP     EL2

CON$INIT:
        int     11h
        and     al,00110000b
        cmp     al,00110000b
        jnz     iscolor
        mov     [base],0b000h           ;look for bw card
iscolor:
        cmp     al,00010000b            ;look for 40 col mode
        ja      setbrk
        mov     [mode],0
        mov     [maxcol],39

setbrk:
        XOR     BX,BX
        MOV     DS,BX
        MOV     BX,BRKADR
        MOV     WORD PTR [BX],OFFSET BREAK
        MOV     WORD PTR [BX+2],CS

        MOV     BX,29H*4
        MOV     WORD PTR [BX],OFFSET COUT
        MOV     WORD PTR [BX+2],CS
```

```
        LDS     BX,CS:[PTRSAV]
        MOV     WORD PTR [BX].TRANS,OFFSET CON$INIT
                                ;SET BREAK ADDRESS
        MOV     [BX].TRANS+2,CS
        JMP     EXIT

CODE    ENDS
        END
```

Chapter 3 MS-DOS Technical Information

3.1 MS-DOS Initialization 3-1
3.2 The Command Processor 3-1
3.3 MS-DOS Disk Allocation 3-2
3.4 MS-DOS Disk Directory 3-3
3.5 File Allocation Table 3-7
3.5.1 How to Use the File Allocation Table 3-8
3.6 MS-DOS Standard Disk Formats 3-9

CHAPTER 3

MS-DOS TECHNICAL INFORMATION

3.1 MS-DOS INITIALIZATION

MS-DOS initialization consists of several steps. Typically, a ROM (Read Only Memory) bootstrap obtains control, and then reads the boot sector off the disk. The boot sector then reads the following files:

IO.SYS
MSDOS.SYS

Once these files are read, the boot process begins.

3.2 THE COMMAND PROCESSOR

The command processor supplied with MS-DOS (file COMMAND.COM.) consists of 3 parts:

1. A resident part resides in memory immediately following MSDOS.SYS and its data area. This part contains routines to process Interrupts 23H (CONTROL-C Exit Address) and 24H (Fatal Error Abort Address), as well as a routine to reload the transient part, if needed. All standard MS-DOS error handling is done within this part of COMMAND.COM. This includes displaying error messages and processing the Abort, Retry, or Ignore messages.

2. An initialization part follows the resident part. During startup, the initialization part is given control; it contains the AUTOEXEC file processor setup routine. The initialization part determines the segment address at which programs can be loaded. It is overlaid by the first program COMMAND.COM loads because it is no longer needed.

3. A transient part is loaded at the high end of memory. This part contains all of the internal command processors and the batch file processor.

   The transient part of the command processor produces the system prompt (such as A>), reads the command from keyboard (or batch file), and causes it to be executed. For external commands, this part builds a command line and issues the EXEC system call (Function Request 4BH) to load and transfer control to the program.

3.3 MS-DOS DISK ALLOCATION

The MS-DOS area is formatted as follows:

Reserved area - variable size

First copy of file allocation table - variable size

Second copy of file allocation table - variable size (optional)

Additional copies of file allocation table - variable size (optional)

Root directory - variable size

File data area

Allocation of space for a file in the data area is not pre-allocated. The space is allocated one cluster at a time. A cluster consists of one or more consecutive sectors; all of the clusters for a file are "chained" together in the File Allocation Table (FAT). (Refer to Section 3.5, "File Allocation Table," for more information on the FAT.) There is usually a second copy of the FAT kept, for consistency. Should the disk develop a bad sector in the middle of the first FAT, the second can be used. This avoids loss of data due to an unusable disk.

3.4 MS-DOS DISK DIRECTORY

FORMAT builds the root directory for all disks. Its location on disk and the maximum number of entries are dependent on the media.

Since directories other than the root directory are regarded as files by MS-DOS, there is no limit to the number of files they may contain.

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

0-7 Filename. Eight characters, left aligned and padded, if necessary, with blanks. The first byte of this field indicates the file status as follows:

- 00H The directory entry has never been used. This is used to limit the length of directory searches, for performance reasons.
- 2EH The entry is for a directory. If the second byte is also 2EH, then the cluster field contains the cluster number of this directory's parent directory (0000H if the parent directory is the root directory). Otherwise, bytes 01H through 0AH are all spaces, and the cluster field contains the cluster number of this directory.
- E5H The file was used, but it has been erased.

Any other character is the first character of a filename.

8-0A Filename extension.

0B File attribute. The attribute byte is mapped as follows (values are in hexadecimal):

- 01 File is marked read-only. An attempt to open the file for writing using the Open File system call (Function Request 3DH) results in an error code being returned. This value can be used along with other values below. Attempts to delete the file with the Delete File system call (13H) or Delete a Directory Entry (41H) will also fail.
- 02 Hidden file. The file is excluded from normal directory searches.

04 System file. The file is excluded from normal directory searches.

08 The entry contains the volume label in the first 11 bytes. The entry contains no other usable information (except date and time of creation), and may exist only in the root directory.

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

20 Archive bit. The bit is set to "on" whenever the file has been written to and closed.

Note: The system files (IO.SYS and MSDOS.SYS) are marked as read-only, hidden, and system files. Files can be marked hidden when they are created. Also, the read-only, hidden, system, and archive attributes may be changed through the Change Attributes system call (Function Request 43H).

0C-15 Reserved.

16-17 Time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:

Offset 17H

H	H	H	H	H	M	M	M
7				3	2		

Offset 16H

M	M	M	S	S	S	S	S
		5	4				0

where:

H is the binary number of hours (0-23)
M is the binary number of minutes (0-59)
S is the binary number of two-second increments

18-19 Date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

Offset 19H

Y	Y	Y	Y	Y	Y	Y	M
7						1	0

Offset 18H

M	M	M	D	D	D	D	D
		5	4				0

where:

Y is 0-119 (1980-2099)
M is 1-12
D is 1-31

1A-1B Starting cluster; the cluster number of the first cluster in the file.

Note that the first cluster for data space on all disks is cluster 002.

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

NOTE

Refer to the Section 3.5.1, "How to Use the File Allocation Table," for details about converting cluster numbers to logical sector numbers.

1C-1F File size in bytes. The first word of this four-byte field is the low-order part of the size.

3.5 FILE ALLOCATION TABLE (FAT)

The following information is included for system programmers who wish to write installable device drivers. This section explains how MS-DOS uses the File Allocation Table to convert the clusters of a file to logical sector numbers. The driver is then responsible for locating the logical sector on disk. Programs must use the MS-DOS file management function calls for accessing files; programs that access the FAT are not guaranteed to be upwardly-compatible with future releases of MS-DOS.

The File Allocation Table is an array of 12-bit entries (1.5 bytes) for each cluster on the disk. The first two FAT entries map a portion of the directory; these FAT entries indicate the size and format of the disk.

The second and third bytes currently always contain FFH.

The third FAT entry, which starts at byte offset 4, begins the mapping of the data area (cluster 002). Files in the data area are not always written sequentially on the disk. The data area is allocated one cluster at a time, skipping over clusters already allocated. The first free cluster found will be the next cluster allocated, regardless of its physical location on the disk. This permits the most efficient utilization of disk space because clusters made available by erasing files can be allocated for new files.

Each FAT entry contains three hexadecimal characters:

000	If the cluster is unused and available.
FF7	The cluster has a bad sector in it. MS-DOS will not allocate such a cluster. CHKDSK counts the number of bad clusters for its report. These bad clusters are not part of any allocation chain.
FF8-FFF	Indicates the last cluster of a file.
XXX	Any other characters that are the cluster number of the next cluster in the file. The cluster number of the first cluster in the file is kept in the file's directory entry.

The File Allocation Table always begins on the first section after the reserved sectors. If the FAT is larger than one sector, the sectors are continguous. Two copies of the FAT are usually written for data integrity. The FAT is read into one of the MS-DOS buffers whenever needed (open, read, write, etc.). For performance reasons, this buffer is given a high priority to keep it in memory as long as possible.

3.5.1 How To Use The File Allocation Table

Use the directory entry to find the starting cluster of the file. Next, to locate each subsequent cluster of the file:

1. Multiply the cluster number just used by 1.5 (each FAT entry is 1.5 bytes long).

2. The whole part of the product is an offset into the FAT, pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.

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

4. If the last cluster used was an even number, keep the low-order 12 bits of the register by ANDing it with FFF; otherwise, keep the high-order 12 bits by shifting the register right 4 bits with a SHR instruction.

5. If the resultant 12 bits are FF8H-FFFH, the file contains no more clusters. Otherwise, the 12 bits contain the cluster number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by DEBUG):

1. Subtract 2 from the cluster number.

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

3. Add to this result the logical sector number of the beginning of the data area.

3.6 MS-DOS STANDARD DISK FORMATS

On an MS-DOS disk, the clusters are arranged on disk to minimize head movement for multi-sided media. All of the space on a track (or cylinder) is allocated before moving on to the next track. This is accomplished by using the sequential sectors on the lowest-numbered head, then all the sectors on the next head, and so on, until all sectors on all heads of the track are used. The next sector to be used will be sector 1 on head 0 of the next track.

For disks, the following can be used:

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

Figure 3.1 5-1/4" Disk Format

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

Table 3.1 MS-DOS Standard Disk Formats

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

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

Chapter 4 MS-DOS Control Blocks and Work Areas

4.1 Typical MS-DOS Memory Map 4-1
4.2 MS-DOS Program Segment 4-2

CHAPTER 4

MS-DOS CONTROL BLOCKS AND WORK AREAS

4.1 TYPICAL MS-DOS MEMORY MAP

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

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

2. User memory is allocated from the lowest end of available memory that will meet the allocation request.

4.2 MS-DOS PROGRAM SEGMENT

When an external command is typed, or when you execute a program through the EXEC system call, MS-DOS determines the lowest available free memory address to use as the start of the program. This area is called the Program Segment.

The first 256 bytes of the Program Segment are set up by the EXEC system call for the program being loaded into memory. The program is then loaded following this block. An .EXE file with minalloc and maxalloc both set to zero is loaded as high as possible.

At offset 0 within the Program Segment, MS-DOS builds the Program Segment Prefix control block. The program returns from EXEC by one of four methods:

1. A long jump to offset 0 in the Program Segment Prefix
2. By issuing an INT 20H with CS:0 pointing at the PSP
3. By issuing an INT 21H with register AH=0 with CS:0 pointing at the PSP, or 4CH and no restrictions on CS
4. By a long call to location 50H in the Program Segment Prefix with AH=0 or Function Request 4CH

NOTE

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

All four methods result in transferring control to the program that issued the EXEC. During this returning process, Interrupts 22H, 23H, and 24H (Terminate Address, CONTROL-C Exit Address, and Fatal Error Abort Address) addresses are restored from the values saved in the Program Segment Prefix of the terminating program. Control is then given to the terminate address. If this is a program returning to COMMAND.COM, control transfers to its resident portion. If a batch file was in process, it is continued;

otherwise, COMMAND.COM performs a checksum on the transient part, reloads it if necessary, then issues the system prompt and waits for you to type the next command.

When a program receives control, the following conditions are in effect:

For all programs:

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

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

NAME=parameter

Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros. The environment built by the command processor contains at least a COMSPEC= string (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings defined with the MS-DOS SET command.

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

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

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

File control blocks at 5CH and 6CH are formatted from the first two parameters typed when the command was entered. If either parameter contained a pathname, then the corresponding FCB contains only the valid drive number. The filename field will not be valid.

An unformatted parameter area at 81H contains all the characters typed after the command (including leading and imbedded delimiters), with the byte at 80H set to the number of characters. If the <, >, or parameters were typed on the command line, they (and the filenames associated with them) will not appear in this area; redirection of standard input and output is transparent to applications.

Offset 6 (one word) contains the number of bytes available in the segment.

Register AX indicates whether or not the drive specifiers (entered with the first two parameters) are valid, as follows:

AL=FF if the first parameter contained an invalid drive specifier (otherwise AL=00)

AH=FF if the second parameter contained an invalid drive specifier (otherwise AH=00)

Offset 2 (one word) contains the segment address of the first byte of unavailable memory. Programs must not modify addresses beyond this point unless they were obtained by allocating memory via the Allocate Memory system call (Function Request 48H).

For Executable (.EXE) programs:

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

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

For Executable (.COM) programs:

All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.

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

The Instruction Pointer (IP) is set to 100H.

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

A word of zeros is placed on top of the stack. This is to allow a user program to exit to COMMAND.COM by doing a RET instruction last. This assumes, however, that the user has maintained his stack and code segments.

Figure 4.1 illustrates the format of the Program Segment Prefix. All offsets are in hexadecimal.

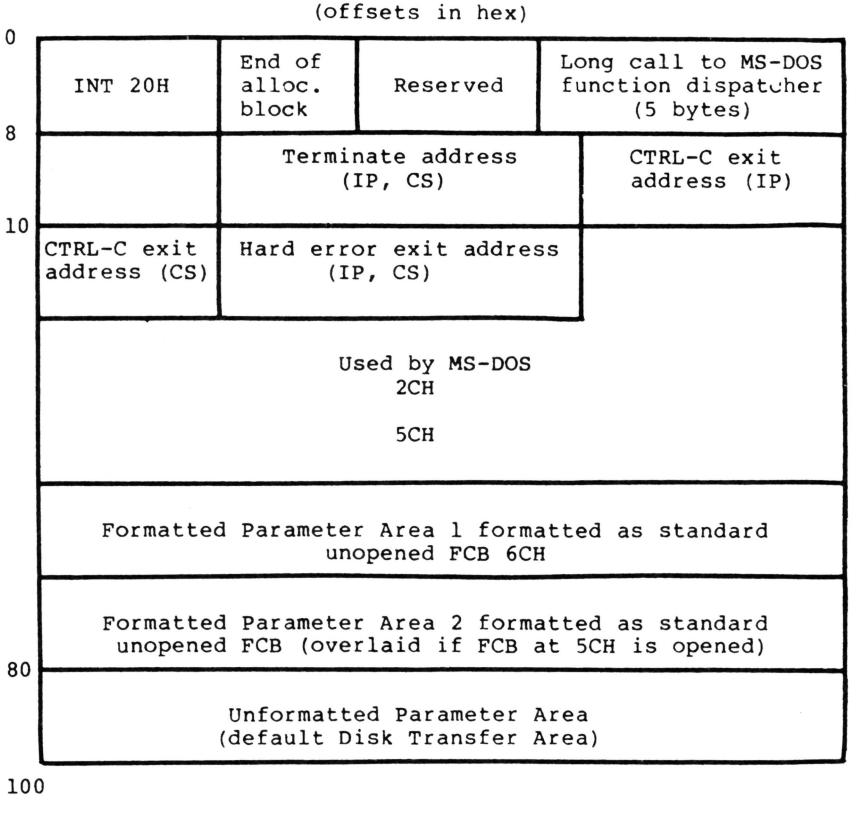


Figure 4.1 Program Segment Prefix

IMPORTANT

Programs must not alter any part of the Program Segment Prefix below offset 5CH.

Chapter 5 EXE File Structure and Loading

CHAPTER 5

.EXE FILE STRUCTURE AND LOADING

NOTE

This chapter describes .EXE file structure and loading procedures for systems that use a version of MS-DOS that is lower than 2.0. For MS-DOS 2.0 and higher, use Function Request 4BH, Load and Execute a Program, to load (or load and execute) an .EXE file.

The .EXE files produced by MS-LINK consist of two parts:

Control and relocation information

The load module

The control and relocation information is at the beginning of the file in an area called the header. The load module immediately follows the header.

The header is formatted as follows. (Note that offsets are in hexadecimal.)

Offset	Contents
00-01	Must contain 4DH, 5AH.
02-03	Number of bytes contained in last page; this is useful in reading overlays.
04-05	Size of the file in 512-byte pages, including the header.
06-07	Number of relocation entries in table.
08-09	Size of the header in 16-byte paragraphs. This is used to locate the beginning of the load module in the file.

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

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

1. The formatted part of the header is read into memory. Its size is 1BH.

2. A portion of memory is allocated depending on the size of the load module and the allocation numbers (0A-0B and 0C-0D). MS-DOS attempts to allocate FFFFH paragraphs. This will always fail, returning the size of the largest free block. If this block

is smaller than minalloc and loadsize, then there will be no memory error. If this block is larger than maxalloc and loadsize, MS-DOS will allocate (maxalloc + loadsize). Otherwise, MS-DOS will allocate the largest free block of memory.

3. A Program Segment Prefix is built in the lowest part of the allocated memory.

4. The load module size is calculated by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted based on the contents of offsets 02-03. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.

5. The load module is read into memory beginning with the start segment.

6. The relocation table items are read into a work area.

7. Each relocation table item segment value is added to the start segment value. This calculated segment, plus the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.

8. Once all relocation items have been processed, the SS and SP registers are set from the values in the header. Then, the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is the initial CS:IP to transfer to before starting execution of the program.

Contents

Chapter 6 Intel Relocatable Object Module Formats

6.1 Introduction 6-1
6.2 Definition of Terms 6-2
6.3 Module Identification and Attributes 6-6
6.4 Segment Definition 6-6
6.5 Segment Addressing 6-6
6.6 Symbol Definition 6-7
6.7 Indices 6-7
6.8 Conceptual Framework for Fixups 6-8
6.9 Self-Relative Fixups 6-14
6.10 Segment-Relative Fixups 6-15
6.15 Record Order 6-15
6.12 Introduction to the Record Formats 6-18
6.13 Numeric List of Record Types 6-50
6.14 Intel Type Representations 6-51
6.15 Microsoft Type Representations for Communal Variables 6-55

CHAPTER 6

INTEL RELOCATABLE OBJECT MODULE FORMATS

6.1 INTRODUCTION

This chapter presents the object record formats that define the object language for the 8086 microprocessor. The 8086 object language is the output of all language translators that have the 8086 as the target processor. The 8086 object language is input and output for object language processors such as linkers and librarians.

The 8086 object module formats permit you to specify relocatable memory images that may be linked together. Capabilities are provided that allow efficient use of the memory mapping facilities of the 8086 microprocessor.

The following table lists the record formats that are supported by Microsoft. These record formats are described in this chapter. Record formats that are preceded by an asterisk (*) deviate from the Intel specification.

Table 6.1 Object Module Record Formats

```
    T-MODULE HEADER RECORD
    LIST OF NAMES RECORD
   *SEGMENT DEFINITION RECORD
   *GROUP DEFINITION RECORD
   *TYPE DEFINITION RECORD

 Symbol Definition Records
   *PUBLIC NAMES DEFINITION RECORD
   *EXTERNAL NAMES DEFINITION RECORD
   *LINE NUMBERS RECORD

  Data Records
    LOGICAL ENUMERATED DATA RECORD
    LOGICAL ITERATED DATA RECORD

    FIXUP RECORD
   *MODULE END RECORD
    COMMENT RECORD
```

6.2 DEFINITION OF TERMS

The following terms are fundamental to the 8086 relocation and linkage.

OMF - Object Module Formats.

MAS - Memory Address Space. The 8086 MAS is 1 megabyte (1,048,576). Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

MODULE - an "inseparable" collection of object code and other information produced by a translator.

T-MODULE - A module created by a translator, such as Pascal or FORTRAN.

The following restrictions apply to object modules:

1. Every module should have a name. Translators will provide names for T-modules, providing a default name (possibly the filename or a null name) if neither source code nor user specifies otherwise.

2. Every T-module in a collection of linked modules must have a different name, so that symbolic debugging systems can distinguish the various line numbers and local symbols. This restriction is not required by the Linker, and is not enforced by it.

FRAME - A contiguous region of 64K of MAS, beginning on a paragraph boundary (i.e., on a multiple of 16 bytes). This concept is useful because the content of the four 8086 segment registers defines four (possibly overlapping) FRAMEs; no 16-bit address in the 8086 code can access a memory location outside of the current four FRAMEs.

LSEG - Logical Segment - A contiguous region of memory whose contents are determined at translation time (except for address-binding). Neither size nor location in MAS are necessarily determined at translation time: size, although partially fixed, may not be final because the LSEG may be combined at LINK time with other LSEGs, forming a single LSEG. An LSEG must not be larger than 64K, so that it can fit in a FRAME. This means that any byte in an LSEG may be addressed by a 16-bit offset from the base of a FRAME covering the LSEG.

PSEG - Physical Segment - This term is equivalent to FRAME. Some people prefer "PSEG" to "FRAME" because the terms "PSEG" and "LSEG" reflect the "physical" and "logical" nature of the underlying segments.

FRAME NUMBER - Every FRAME begins on a paragraph boundary. The "paragraphs" in MAS can be numbered from 0 through 65535. These numbers, each of which defines a FRAME, are called FRAME NUMBERS.

PARAGRAPH NUMBER - This term is equivalent to "FRAME NUMBER."

PSEG NUMBER - This term is equivalent to "FRAME NUMBER."

GROUP - A collection of LSEGs defined at translation time, whose final locations in MAS have been constrained such that there will be at least one FRAME which covers (contains) every LSEG in the collection.

The notation "Gr A(X,Y,Z,)" means that LSEGs X, Y and Z form a group whose name is A. The fact that X, Y and Z are all LSEGs in the same group does not imply any ordering of X, Y and Z in MAS, nor does it imply any contiguity between X, Y and Z.

The Microsoft Linker does not currently allow an LSEG to be a member of more than one group. The Linker will ignore all attempts to place an LSEG in more than one group.

CANONIC - Any location in MAS is contained in exactly 4096 distinct FRAMEs; but one of these FRAMEs can be distinguished because it has a higher FRAME NUMBER. This distinguished FRAME is called the canonic FRAME of the location. In other words, the canonic frame of a given byte is the frame so chosen that the byte's offset from that frame lies in the range 0 to 15 (decimal). Thus, if FOO is a symbol defining a memory location, one may speak of the "canonic FRAME of FOO", or of "FOO's canonic FRAME". By extension, if S is any set of memory locations, then there exists a unique FRAME which has the lowest FRAME NUMBER in the set of canonic FRAMEs of the locations in S. This unique FRAME is called the canonic FRAME of the set S. Thus, we may speak of the canonic FRAME of an LSEG or of a group of LSEGs.

SEGMENT NAME - LSEGs are assigned segment names at translation time. These names serve two purposes:

1. They play a role at LINK time in determining which LSEGs are combined with other LSEGs.

2. They are used in assembly source code to specify groups.

CLASS NAME - LSEGs may optionally be assigned Class Names at translation time. Classes define a partition on LSEGs: two LSEGs are in the same class if they have the same Class Name.

The Microsoft Linker applies the following semantics to class names. The class name "CODE" or any class name whose suffix is "CODE" implies that all segments of said class contain only code and may be considered read-only. Such segments may be overlayed if the user specifies the module containing the segment as part of an overlay.

OVERLAY NAME - LSEGs may optionally be assigned an overlay name. The overlay name of an LSEG is ignored by MS-LINK (version 2.40 and later versions), but it is used by Intel relocation and Linkage products.

COMPLETE NAME - The complete name of an LSEG consists of the Segment Name, Class Name, and Overlay Name. LSEGs from different modules will be combined if their Complete Names are identical.

6.3 MODULE IDENTIFICATION AND ATTRIBUTES

A module header record is always the first record in a module. It provides a module name.

In addition to a name, a module may have the attribute of being a main program as well as having a specified starting address. When linking multiple modules together, only one module with the main attribute should be given.

In summary, modules may or may not be main and may or may not have a starting address.

6.4 SEGMENT DEFINITION

A module is a collection of object code defined by a sequence of records produced by a translator. The object code represents contiguous regions of memory whose contents are determined at translation time. These regions are called LOGICAL SEGMENTS (LSEGs). A module defines the attributes of each LSEG. The SEGMENT DEFINITION RECORD (SEGDEF) is the vehicle by which all LSEG information (name, length, memory alignment, etc.) is maintained. The LSEG information is required when multiple LSEGs are combined and when segment addressability (See Section 6.5, "Segment Addressing") is established. The SEGDEF records are required to follow the first header record.

6.5 SEGMENT ADDRESSING

The 8086 addressing mechanism provides segment base registers from which a 64K-byte region of memory, called a FRAME, may be addressed. There is one code segment base register (CS), two data segment base registers (DS, ES), and one stack segment base register (SS).

The possible number of LSEGs that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would be the case in a modular program with many small data and/or code LSEGs.

Since such frequent loading of base registers is undesirable, it is a good strategy to collect many small LSEGs together into a single unit that will fit in one memory frame so that all the LSEGs may be addressed using the same base register value. This addressable unit is a

GROUP and has been defined earlier in the DEFINITION OF TERMS.

To allow addressability of objects within a GROUP to be established, each GROUP must be explicitly defined in the module. The GROUP DEFINITION RECORD (GRPDEF) provides a list of constituent segments either by segment name or by segment attribute such as "the segment defining symbol FOO" or "the segments with class name ROM."

The GRPDEF records within a module must follow all SEGDEF records as GRPDEF records may reference SEGDEF records in defining a GROUP. The GRPDEF records must also precede all other records except header records, as the Linker must process them first.

6.6 SYMBOL DEFINITION

MS-LINK supports three different types of records that fall into the class of symbol definition records. The two most important types are PUBLIC NAMES DEFINITION RECORDs (PUBDEFs) and EXTERNAL NAMES DEFINITION RECORDS (EXTDEFs). These types are used to define globally visible procedures and data items and to resolve external references. In addition, TYPDEF records are used by MS-LINK for the allocation of communal variables (see Section 6.15 "Microsoft Type Representations for Communal Variables".)

6.7 INDICES

"Index" fields occur throughout this document. An index is an integer that selects some particular item from a collection of such items. (List of examples: NAME INDEX, SEGMENT INDEX, GROUP INDEX, EXTERNAL INDEX, TYPE INDEX.)

NOTE

An index is normally a positive number. The index value zero is reserved, and may carry a special meaning dependent upon the type of index (e.g., a Segment Index of zero specifies the "Unnamed," absolute pseudo-segment; a Type Index of zero specifies the "Untyped type", which is different from "Decline to state").

In general, indices must assume values quite large (that is, much larger than 255). Nevertheless, a great number of object files will contain no indices with values greater than 50 or 100. Therefore, indices will be encoded in 1 or 2 bytes, as required.

The high-order (left-most) bit of the first (and possibly the only) byte determines whether the index occupies one byte or two. If the bit is 0, then the index is a number between 0 and 127, occupying one byte. If the bit is 1, then the index is a number between 0 and 32K-1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.

6.8 CONCEPTUAL FRAMEWORK FOR FIXUPS

A "fixup" is some modification to object code, requested by a translator, performed by the Linker, achieving address binding.

NOTE

This definition of "fixup" accurately represents the viewpoint maintained by the Linker. Nevertheless, the Linker can be used to achieve modifications of object code (i.e., "fixups") that do not conform to this definition. For example, the binding of code to either hardware floating point or software floating point subroutines is a modification to an operation code, where the operation code is treated as if it were an address. The previous definition of "fixup" is not intended to disallow or disparage object code modifications.

8086 translators specify a fixup by giving four data:

1. The place and type of a LOCATION to be fixed up

2. One of two possible fixup MODEs

3. A TARGET, which is a memory address to which LOCATION must refer

4. A FRAME defining a context within which the reference takes place

LOCATION - There are 5 types of LOCATION: a POINTER, a BASE, an OFFSET, a HIBYTE, and a LOBYTE.

The vertical alignment of the following figure illustrates four points. (Remember that the high-order byte of a word in 8086 memory is the byte with the higher address.)

1. A BASE is the high-order word of a pointer (and the Linker doesn't care if the low-order word of the pointer is present or not).

2. An OFFSET is the low-order word of a pointer (and the Linker doesn't care if the high-order word follows or not).

3. A HIBYTE is the high-order half of an OFFSET (and the Linker doesn't care if the low-order half precedes or not).

4. A LOBYTE is the low-order half of an OFFSET (and the Linker doesn't care if the high-order half follows or not).

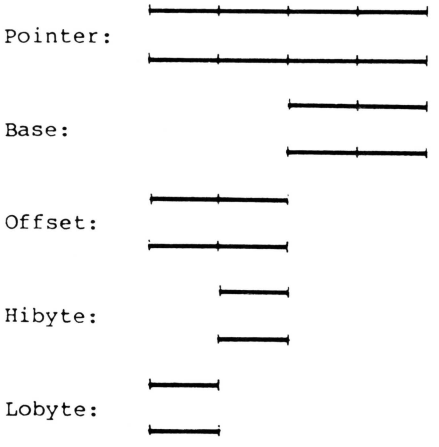


Figure 6.1. LOCATION Types

A LOCATION is specified by two data: (1) the LOCATION type, and (2) where the LOCATION is. The first is specified by the LOC subfield of the LOCAT field of the FIXUP record; the second is specified by the DATA RECORD OFFSET subfield of the LOCAT field of the FIXUP record.

MODE - The Linker supports two kinds of fixups: "self-relative" and "segment-relative."

Self-relative fixups support the 8- and 16-bit offsets that are used in the CALL, JUMP and SHORT-JUMP instructions. Segment-relative fixups support all other addressing modes of the 8086.

TARGET - The TARGET is the location in MAS being referenced. (More explicitly, the TARGET may be considered to be the lowest byte in the object being referenced.) A TARGET is specified in one of eight ways. There are four "primary" ways, and four "secondary" ways. Each primary way of specifying a TARGET uses two kinds of data: an INDEX-or-FRAME-NUMBER 'X', and a displacement 'D'.

(T0) X is a SEGMENT INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.

(T1) X is a GROUP INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.

(T2) X is an EXTERNAL INDEX. The TARGET is the Dth byte following the byte whose address is (eventually) given by the External Name identified by the INDEX.

(T3) X is a FRAME NUMBER. The TARGET is the Dth byte in the FRAME identified by the FRAME NUMBER (i.e., the address of TARGET is (X*16)+D).

Each secondary way of specifying a TARGET uses only one data item: the INDEX-or-FRAME-NUMBER X. An implicit displacement equal to zero is assumed.

(T4) X is a SEGMENT INDEX. The TARGET is the 0th (first) byte in the LSEG identified by the INDEX.

(T5) X is a GROUP INDEX. The TARGET is the 0th (first) byte in the LSEG in the specified group that is eventually LOCATEd lowest in MAS.

(T6) X is an EXTERNAL INDEX. The TARGET is the byte whose address is the External Name identified by the INDEX.

(T7) X is a FRAME NUMBER. The TARGET is the byte whose 20-bit address is (X*16).

NOTE

The Microsoft Linker does not support methods T3 and T7.

The following nomenclature is used to describe a TARGET:

TARGET:	SI(<segment name>), <displacement>	[T0]
TARGET:	GI(<group name>), <displacement>	[T1]
TARGET:	EI(<symbol name>), <displacement>	[T2]
TARGET:	SI (<segment name>)	[T4]
TARGET:	GI (<group name>)	[T5]
TARGET:	EI (<symbol name>)	[T6]

The following examples illustrate how this notation is used:

TARGET:	SI(CODE), 1024	The 1025th byte in the segment "CODE"
TARGET:	GI(DATAAREA)	The location in MAS of a group called "DATAAREA"
TARGET:	EI(SIN)	The address of the external subroutine "SIN"
TARGET:	EI(PAYSCHEDULE), 24	The 24th byte following the location of an EXTERNAL data structure called "PAYSCHEDULE"

FRAME - Every 8086 memory reference is to a location contained within some FRAME; where the FRAME is designated by the content of some segment register. For the Linker to form a correct, usable memory reference, it must know what the TARGET is, and to which FRAME the reference is being made. Thus, every fixup specifies such a FRAME, in one of six ways. Some ways use data, X, which is in INDEX-or-FRAME-NUMBER, as above. Other ways require no data.

The six ways of specifying frames are:

(F0) X is a SEGMENT INDEX. The FRAME is the canonic FRAME of the LSEG defined by the INDEX.

(F1) X is a GROUP INDEX. The FRAME is the canonic FRAME defined by the group (i.e., the canonic FRAME defined by the LSEG in the group that is eventually LOCATEd lowest in MAS).

(F2) X is an EXTERNAL INDEX. The FRAME is determined when the External Name's public definition is found. There are 3 cases:

(F2a) The symbol is defined relative to some LSEG, and there is no associated GROUP. The LSEGs canonic FRAME is specified.

(F2b) The symbol is defined absolutely, without reference to an LSEG, and there is no associated GROUP. The FRAME is specified by the FRAME NUMBER subfield of the PUBDEF record that gives the symbol's definition.

(F2c) Regardless of how the symbol is defined, there is an associated GROUP. The canonic FRAME of the GROUP is specified. (The group is specified by the GROUP INDEX subfield of the PUBDEF Record.)

(F3) X is a FRAME NUMBER (specifying the obvious FRAME).

(F4) No X. The FRAME is the canonic FRAME of the LSEG containing LOCATION.

(F5) No X. The FRAME is determined by the TARGET. There are four cases:

(F5a) The TARGET specified a SEGMENT INDEX: in this case, the FRAME is determined as in (F0).

(F5b) The TARGET specified a GROUP INDEX: in this case, the FRAME is determined as in (F1).

(F5c) The TARGET specified an EXTERNAL INDEX: in this case, the FRAME is determined as in (F2).

(F5d) The TARGET is specified with an explicit FRAME NUMBER: in this case the FRAME is determined as in (F3).

NOTE

The Microsoft Linker does not support frame methods F2b, F3, and F5d.

Nomenclature describing FRAMEs is similar to the above nomenclature for TARGETs.

FRAME:	SI (<segment name>)	[F0]
FRAME:	GI (<group name>)	[F1]
FRAME:	EI (<symbol name>)	[F2]
FRAME:	LOCATION	[F4]
FRAME:	TARGET	[F5]
FRAME:	NONE	[F6]

For an 8086 memory reference, the FRAME specified by a self-relative reference is usually the canonic FRAME of the LSEG containing the LOCATION, and the FRAME specified by a segment relative reference is the canonic FRAME of the LSEG containing the TARGET.

6.9 SELF-RELATIVE FIXUPS

A self-relative fixup operates as follows: A memory address is implicitly defined by LOCATION; namely the address of the byte following LOCATION (because at the time of a self-relative reference, the 8086 IP (Instruction Pointer) is pointing to the byte following the reference).

For 8086 self-relative references, if either LOCATION or TARGET are outside the specified FRAME, the Linker gives a warning. Otherwise, there is a unique 16-bit displacement which, when added to the address implicitly defined by LOCATION, will yield the relative position of TARGET in the FRAME.

If the LOCATION is an OFFSET, the displacement is added to LOCATION modulo 65536; no errors are reported.

If the LOCATION is a LOBYTE, the displacement must be within the range {-128:127}, otherwise the Linker will give a warning. The displacement is added to LOCATION modulo 256.

If the LOCATION is a BASE, POINTER, or HIBYTE, it is unclear what the translator had in mind, and the action taken by the Linker is undefined.

6.10 SEGMENT-RELATIVE FIXUPS

A segment-relative fixup operates in the following way: a non-negative 16-bit number, FBVAL, is defined as the FRAME NUMBER of the FRAME specified by the fixup, and a signed 20-bit number, FOVAL, is defined as the distance from the base of the FRAME to the TARGET. If this signed 20-bit number is less than 0 or greater than 65535, the Linker reports an error. Otherwise, FBVAL and FOVAL are used to fixup LOCATION in the following fashion:

1. If LOCATION is a POINTER, then FBVAL is added (modulo 65536) to the high-order word of POINTER, and FOVAL is added (modulo 65536) to the low-order word of POINTER.

2. If LOCATION is a BASE, then FBVAL is added (modulo 65536) to the BASE; FOVAL is ignored.

3. If LOCATION is an OFFSET, then FOVAL is added (modulo 65536) to the OFFSET; FBVAL is ignored.

4. If LOCATION is a HIBYTE, then (FOVAL/256) is added (modulo 256) to the HIBYTE; FBVAL is ignored. (The indicated division is "integer division", i.e., the remainder is discarded.)

5. If LOCATION is a LOBYTE, then (FOVAL modulo 256) is added (modulo 256) to the LOBYTE; FBVAL is ignored.

6.11 RECORD ORDER

A object code file must contain a sequence of (one or more) modules, or a library containing zero or more modules. A module is defined as a collection of object code defined by a sequence of object records. The following syntax shows

the valid orderings of records to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence.

NOTE

The syntactic description language used below is defined in WIRTH: CACM, November 1977, vol. 20, no. 11, pp. 822-823. The character strings represented by capital letters above are not literals but are identifiers that are further defined in the section describing the record formats.

```
object file   = tmodule

tmodule       = THEADR seg-grp   {component}   modtail

seg_grp       = {LNAMES} {SEGDEF} {TYPDEF | EXTDEF | GRPDEF}

component     = data | debug_record

data          = content_def | thread_def |
                     TYPDEF | PUBDEF | EXTDEF

debug_record  = LINNUM

content_def   = data_record {FIXUPP}

thread_def    = FIXUPP (containing only thread fields)

data_record   = LIDATA | LEDATA

modtail       = MODEND
```

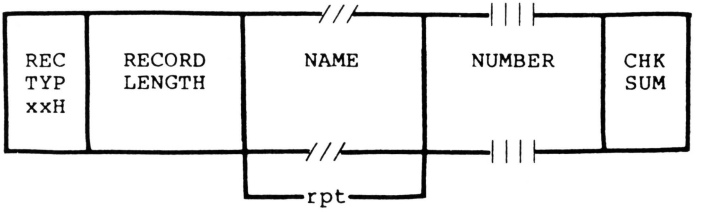
The following rules apply:

1. A FIXUPP record always refers to the previous DATA record.
2. All LNAMES, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.
3. COMENT records may appear anywhere in a file, except as the first or last record in a file or module, or within a contentdef.

6.12 INTRODUCTION TO THE RECORD FORMATS

The following pages present diagrams of record formats in schematic form. Here is a sample record format, to illustrate the various conventions.

SAMPLE RECORD FORMAT
(SAMREC)



TITLE and OFFICIAL ABBREVIATION

At the top is the name of the record format described, with an official abbreviation. To promote uniformity among various programs, including translators and debuggers, the abbreviation should be used in both code and documentation. The record format abbreviation is always six letters.

The BOXES

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes represent two bytes each. The wide boxes with three slashes in the top and bottom represent a variable number of bytes, one or more, depending upon content. The wide boxes with four vertical bars in the top and bottom represent 4-byte fields.

REC TYP

The first byte in each record contains a value between 0 and 255, indicating which record type the record is.

RECORD LENGTH

The second field in each record contains the number of bytes in the record, exclusive of the first two fields.

NAME

Any field that indicates a "NAME" has the following internal structure: the first byte contains a number between 0 and 127, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string.

Most translators constrain the character set to be a subset of the ASCII character set.

NUMBER

A 4-byte NUMBER field represents a 32-bit unsigned integer, where the first 8 bits (least-significant) are stored in the first byte (lowest address), the next 8 bits are stored in the second byte, and so on.

REPEATED OR CONDITIONAL FIELDS

Some portions of a record format contain a field or a series of fields that may be repeated one or more times. Such portions are indicated by the "repeated" or "rpt" brackets below the boxes.

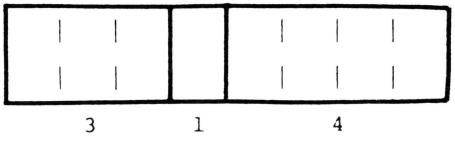
Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar "conditional" or "cond" brackets below the boxes.

CHK.SUM

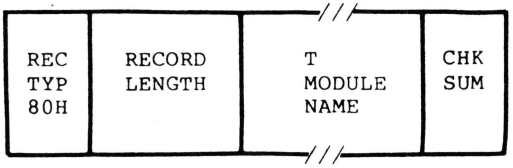
The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

BIT FIELDS

Descriptions of contents of fields will sometimes be at the bit level. Boxes with vertical lines drawn through them represent bytes or words; the vertical lines indicate bit boundaries; thus the byte represented below, has three bit-fields of 3-, 1-, and 4-bits.



T-MODULE HEADER RECORD
(THEADR)

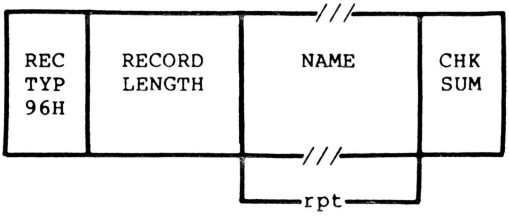


Every module output from a translator must have a T-MODULE HEADER RECORD.

T-MODULE NAME

The T-MODULE NAME provides a name for the T-MODULE.

LIST OF NAMES RECORD (LNAMES)



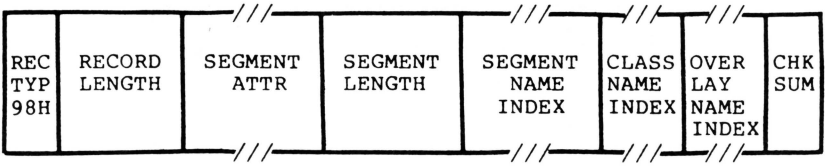
This Record provides a list of names that may be used in following SEGDEF and GRPDEF records as the names of Segments, Classes and/or Groups.

The ordering of LNAMES records within a module, together with the ordering of names within each LNAMES Record, induces an ordering on the names. Thus, these names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as "Name Indices" in the Segment Name Index, Class Name Index and Group Name Index fields of the SEGDEF and GRPDEF Records.

NAME

This repeatable field provides a name, which may have zero length.

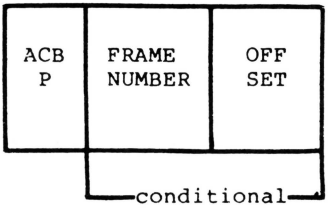
SEGMENT DEFINITION RECORD (SEGDEF)



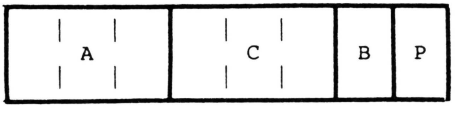
SEGMENT INDEX values 1 through 32767, which are used in other record types to refer to specific LSEGs, are defined implicitly by the sequence in which SEGDEF Records appear in the object file.

SEG ATTR

The SEG ATTR field provides information on various attributes of a segment, and has the following format:



The ACBP byte contains four numbers which are the A, C, B, and P attribute specifications. This byte has the following format:



A (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

A=0 SEGDEF describes an absolute LSEG.
A=1 SEGDEF describes a relocatable, byte-aligned LSEG.
A=2 SEGDEF describes a relocatable, word-aligned LSEG.
A=3 SEGDEF describes a relocatable, paragraph-aligned LSEG.
A=4 SEGGDEF describes a relocatable, page-aligned LSEG.

If A=0, the FRAME NUMBER and OFFSET fields will be present. Using MS-LINK, absolute segments may be used for addressing purposes only; for example, defining the starting address of a ROM and defining symbolic names for addresses within the ROM. MS-LINK will ignore any data specified as belonging to an absolute LSEG.

C (Combination) is a 3-bit subfield that specifies the combination attribute of the LSEG. Absolute segments (A=0) must have combination zero (C=0). For relocatable segments, the C field encodes a number (0,1,2,4,5,6 or 7) that indicates how the segment can be combined. The interpretation of this attribute is best given by considering how two LSEGs are combined: Let X,Y be LSEGs, and let Z be the LSEG resulting from the combination of X,Y. Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX, LY. Let G be the length of any gap required between the X- and Y-components of Z to accommodate the alignment attribute of Y. Let LZ denote the length of the (combined) LSEG Z; let dx (0<=dx<LX) be the offset in X of a byte, and let dy similarly be the offset in Y of a byte. The following table gives the length LZ of the combined LSEG Z, and the offsets dx' and dy' in Z for the bytes corresponding to dx in X and dy in Y.

Table 6.2. Combination Attribute Example

C	LZ	dx'	dy'	
2	LX+LY+G	dx	dy+LX+G	"Public"
5	LX+LY+G	dx	dy+LX+G	"Stack"
6	MXY	dx	dy	"Common"

Table 6.2 has no lines for C=0, C=1, C=3, C=4 and C=7. C=0 indicates that the relocatable LSEG may not be combined; C=1 and C=3 are undefined. C=4 and C=7 are treated like C=2.

B (Big) is a 1-bit subfield which, if 1, indicates that the Segment Length is exactly 64K (65536). In this case the SEGMENT LENGTH field must contain zero.

The P field must always be zero.

The FRAME NUMBER and OFFSET fields (present only for absolute segments, A=0) specify the placement in MAS of the absolute segment. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET, then an adjustment of the FRAME NUMBER should be done.

SEGMENT LENGTH

The SEGMENT LENGTH field gives the length of the segment in bytes. The length may be zero; if so, MS-LINK will not delete the segment from the module. The SEGMENT LENGTH field is only big enough to hold numbers from 0 to 64K-1 inclusive. The B attribute bit in the ACBP field (see SEG ATTR section) must be used to give the segment a length of 64K.

SEGMENT NAME INDEX

The Segment Name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, TAXDATA, MODULENAME_CODE, STACK. This field provides the Segment Name, by indexing into the list of names provided by the LNAMES Record(s).

CLASS NAME INDEX

The Class Name is a name the programmer or translator can assign to a segment. If none is assigned, the name is null, and has length 0. The purpose of Class Names is to allow the programmer to define a "handle" used in the ordering of the LSEGs in MAS. Examples: RED, WHITE, BLUE; ROM FASTRAM, DISPLAYRAM. This field provides the Class Name, by indexing into the list of names provided by the LNAMES Record(s).

OVERLAY NAME INDEX

NOTE

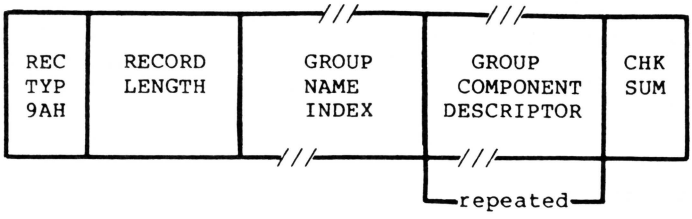
This is ignored in MS-LINK versions 2.40 and later, but supported in all earlier versions. However, semantics differ from Intel semantics.

The Overlay Name is a name the translator and/or MS-LINK, at the programmer's request, applies to a segment. The Overlay Name, like the Class Name, may be null. This field provides the Overlay Name, by indexing into the list of names provided by the LNAMES Record(s).

NOTE

The "Complete Name" of a segment is a 3-component entity comprising a Segment Name, a Class Name and an Overlay Name. (The latter two components may be null.)

GROUP DEFINITION RECORD
(GRPDEF)



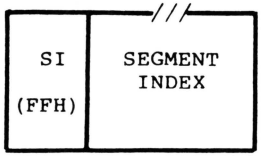
GROUP NAME INDEX

The Group Name is a name by which a collection of LSEGs may be referenced. The important property of such a group is that, when the LSEGs are eventually fixed in MAS, there must exist some FRAME which "covers" every LSEG of the group.

The GROUP NAME INDEX field provides the Group Name, by indexing into the list of names provided by the LNAMES Record(s).

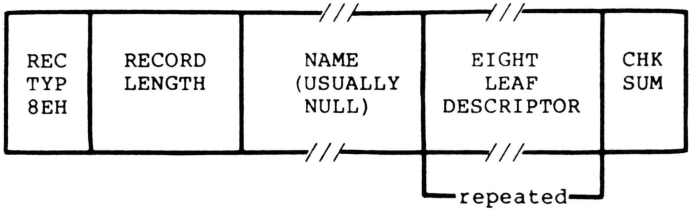
GROUP COMPONENT DESCRIPTOR

Each GROUP COMPONENT DESCRIPTOR has the following format:



The first byte of the DESCRIPTOR contains 0FFH; the DESCRIPTOR contains one field, which is a SEGMENT INDEX that selects the LSEG described by a preceding SEGDEF record.

TYPE DEFINITION RECORD
(TYPDEF)



The Microsoft Linker uses TYPDEF records only for communal variable allocation. This is not Intel's intended purpose. See Section 6.15, "Microsoft Type Representations for Communal Variables."

As many "EIGHT LEAF DESCRIPTOR" fields as necessary are used to describe a branch. (Every such field except the last in the record describes eight leaves; the last such field describes from one to eight leaves.)

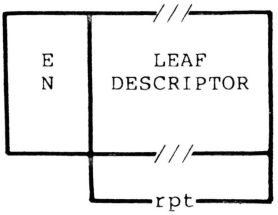
TYPE INDEX values 1 through 32767, which are contained in other record types to associate object types with object names, are defined implicitly by the sequence in which TYPDEF records appear in the object file.

NAME

Use of this field is reserved. Translators should place a single byte containing 0 in it (which is the representation of a name of length zero).

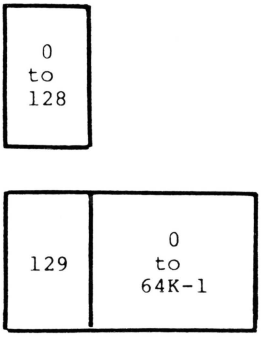
EIGHT LEAF DESCRIPTOR

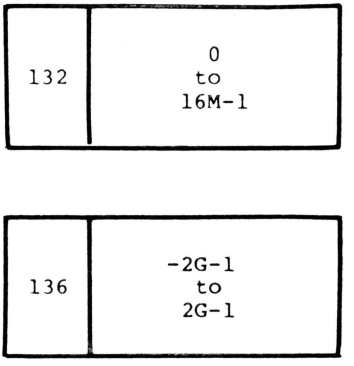
This field can describe up to eight Leaves.



The EN field is a byte: the 8 bits, left to right, indicate if the following 8 Leaves (left to right) are Easy (bit=0) or Nice (bit=1).

The LEAF DESCRIPTOR field, which occurs between 1 and 8 times, has one of the following formats:





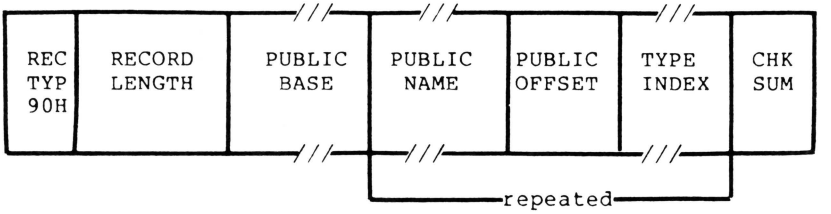
The first format (single byte), containing a value between 0 and 127, represents a Numeric Leaf whose value is the number given.

The second format, with a leading byte containing 129, represents a Numeric Leaf. The number is contained in the following two bytes.

The third format, with a leading byte containing 132, represents a Numeric Leaf. The number is contained in the following three bytes.

The fourth format, with a leading byte containing 136, represents a Signed Numeric Leaf. The number is contained in the following four bytes, sign extended if necessary.

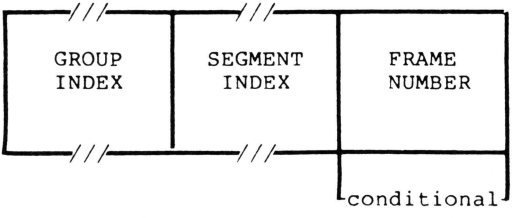
PUBLIC NAMES DEFINITION RECORD
(PUBDEF)



This record provides a list of one or more PUBLIC NAMEs; for each one, three data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

PUBLIC BASE

The PUBLIC BASE has the following format:



The GROUP INDEX field has a format given earlier, and provides a number between 0 and 32767 inclusive. A non-zero GROUP INDEX associates a group with the public symbol, and is used as described in Section 6.8, "Conceptual Framework for Fixups," case (F2c). A zero GROUP INDEX indicates that there is no associated group.

The SEGMENT INDEX field has a format given earlier, and provides a number between 0 and 32767, inclusive.

A non-zero SEGMENT INDEX selects an LSEG. In this case, the location of each public symbol defined in the record is taken as a non-negative displacement (given by a PUBLIC OFFSET field) from the first byte of the selected LSEG, and the FRAME NUMBER field must be absent.

A SEGMENT INDEX of 0 (legal only if GROUP INDEX is also 0) means that the location of each public symbol defined in the record is taken as a displacement from the base of the FRAME defined by the value in the FRAME NUMBER field.

The FRAME NUMBER is present if both the SEGMENT INDEX and GROUP INDEX are zero.

A non-zero GROUP INDEX selects some group; this group is taken as the "frame of reference" for references to all public symbols defined in this record; that is, MS-LINK will perform the following actions:

1. Any fixup of the form:

   TARGET: EI(P)

   FRAME: TARGET

   (where "P" is a public symbol in this PUBDEF record) will be converted by MS-LINK to a fixup of the form:

   TARGET: SI(L),d

   FRAME: GI(G)

   where "SI(L)" and "d" are provided by the SEGMENT INDEX and PUBLIC OFFSET fields. (The "normal" action would have the frame specifier in the new fixup be the same as in the old fixup: FRAME: TARGET.)

2. When the value of a public symbol, as defined by the SEGMENT INDEX, PUBLIC OFFSET, and (optionally) FRAME NUMBER fields, is converted to a {base,offset} pair, the base part will be taken as the base of the indicated group. If a non-negative 16-bit offset cannot then complete the definition of the public symbol's value, an error occurs.

A GROUP INDEX of zero selects no group. MS-LINK will not alter the FRAME specification of fixups referencing the symbol, and will take, as the base part of the absolute value of the public symbol, the canonic frame of the segment (either LSEG or PSEG) determined by the SEGMENT INDEX field.

PUBLIC NAME

The PUBLIC NAME field gives the name of the object whose location in MAS is made available to other modules. The name must contain one or more characters.

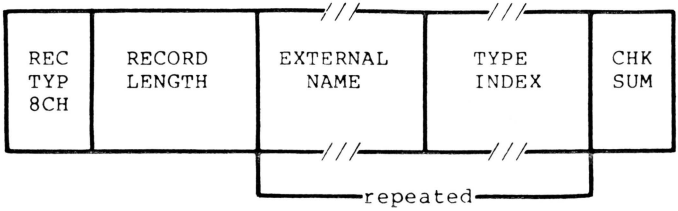
PUBLIC OFFSET

The PUBLIC OFFSET field is a 16-bit value, which is either the offset of the Public Symbol with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the Public Symbol with respect to the specified FRAME (if SEGMENT INDEX = 0).

TYPE INDEX

The TYPE INDEX field identifies a single preceding TYPDEF (Type Definition) Record containing a descriptor for the type of entity represented by the Public Symbol. This field is ignored by the Linker.

EXTERNAL NAMES DEFINITION RECORD
(EXTDEF)



This record provides a list of external names, and for each name, the type of object it represents. MS-LINK will assign to each External Name the value provided by an identical Public Name (if such a name is found).

EXTERNAL NAME

This field provides the name, which must have non-zero length, of an external object.

Inclusion of a Name in an External Names Record is an implicit request that the object file be linked to a module containing the same name declared as a Public Symbol. This request obtains whether or not the External Name is referenced within some FIXUPP Record in the module.

The ordering of EXTDEF Records within a module, together with the ordering of External Names within each EXTDEF Record, induces an ordering on the set of all External Names requested by the module. Thus, External Names are considered to be numbered 1, 2, 3, 4, ... These numbers are used as "External Indices" in the TARGET DATUM and/or FRAME DATUM fields of FIXUPP Records to refer to a particular External Name.

NOTE

8086 External Names are numbered positively: 1,2,3,... This is a change from 8080 External Names, which were numbered starting from zero: 0,1,2,... This conforms with other 8086 Indices (Segment Index, Type Index, etc.) which use 0 as a default value with special meaning.

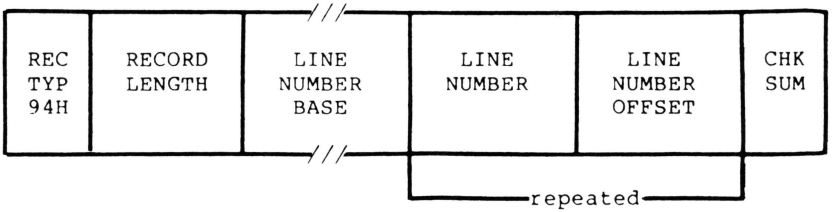
External indices may not reference forward. For example, an external definition record defining the kth object must precede any record referring to that object with index k.

TYPE INDEX

This field identifies a single preceding TYPDEF (Type Definition) record containing a descriptor for the type of object named by the External Symbol.

The TYPE INDEX is used only in communal variable allocation by the Microsoft Linker.

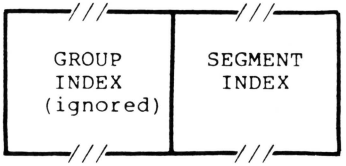
LINE NUMBERS RECORD
(LINNUM)



This record provides the means by which a translator may pass the correspondence between a line number in source code and the corresponding translated code.

LINE NUMBER BASE

The LINE NUMBER BASE has the following format:



The SEGMENT INDEX determines the location of the first byte of code corresponding to some source line number.

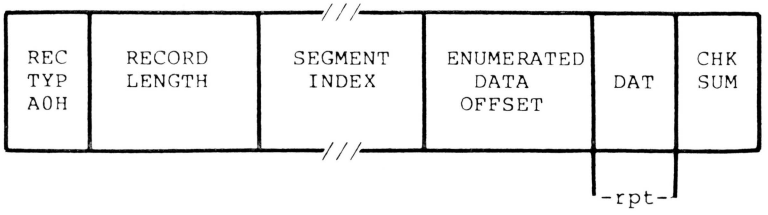
LINE NUMBER

A line number between 0 and 32767, inclusive, is provided in binary by this field. The high-order bit is reserved for future use and must be zero.

LINE NUMBER OFFSET

The LINE NUMBER OFFSET field is a 16-bit value, which is the offset of the line number with respect to an LSEG (if SEGMENT INDEX > 0).

LOGICAL ENUMERATED DATA RECORD
(LEDATA)



This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

SEGMENT INDEX

This field must be non-zero and specifies an index relative to the SEGMENT DEFINITION RECORDS found previous to the LEDATA RECORD.

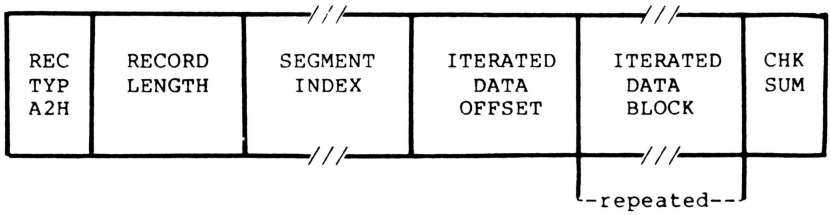
ENUMERATED DATA OFFSET

This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory.

DAT

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

LOGICAL ITERATED DATA RECORD
(LIDATA)



This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

SEGMENT INDEX

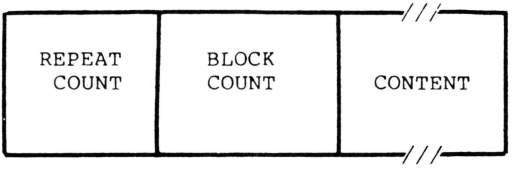
This field must be non-zero and specifies an index relative to the SEGDEF records found previous to the LIDATA RECORD.

ITERATED DATA OFFSET

This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory.

ITERATED DATA BLOCK

This repeated field is a structure specifying the repeated data bytes. The structure has the following format:



NOTE

The Linker cannot handle LIDATA records whose ITERATED DATA BLOCK is larger than 512 bytes.

REPEAT COUNT

This field specifies the number of times that the CONTENT portion of this ITERATED DATA BLOCK is to be repeated. REPEAT COUNT must be non-zero.

BLOCK COUNT

This field specifies the number of ITERATED DATA BLOCKS that are to be found in the CONTENT portion of this ITERATED DATA BLOCK. If this field has value zero, then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as data bytes. If non-zero, then the CONTENT portion is interpreted as that number of ITERATED DATA BLOCKs.

CONTENT

This field may be interpreted in one of two ways, depending on the value of the previous BLOCK COUNT field.

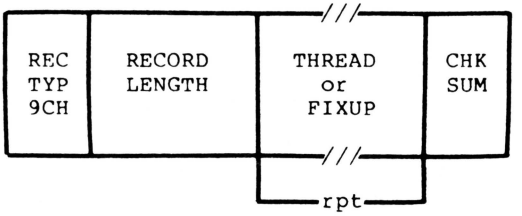
If BLOCK COUNT is zero then this field is a 1-byte count followed by the indicated number of data bytes.

If BLOCK COUNT is non-zero then this field is interpreted as the first byte of another ITERATED DATA BLOCK.

NOTE

From the outermost level, the number of nested ITERATED DATA BLOCKS is limited to 17, i.e., the number of levels of recursion is limited to 17.

FIXUP RECORD
(FIXUPP)

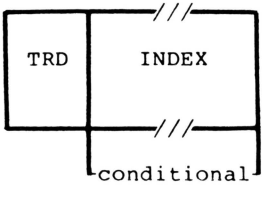


This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a LOCATION within the previous DATA record. A data record may be followed by more than one fixup record that refers. Each fixup is specified by a FIXUP field that specifies 4 data: a location, a mode, a target and a frame. The frame and the target may be specified totally within the FIXUP field, or may be specified by reference to a preceding THREAD field.

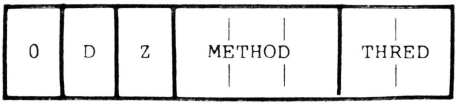
A THREAD field specifies a default target or frame that may subsequently be referred to in identifying a target or a frame. Eight threads are provided; four for frame specification and four for target specification. Once a target or frame has been specified by a THREAD, it may be referred to by following FIXUP fields (in the same or following FIXUPP records), until another THREAD field with the same type (TARGET or FRAME) and Thread Number (0 - 3) appears (in the same or another FIXUPP record).

THREAD

THREAD is a field with the following format:



The TRD DAT (ThReaD DATa) subfield is a byte with this internal structure:



The 'Z' is a 1-bit subfield, currently without any defined function, that is required to contain 0.

The 'D' subfield is one bit that identifies what type of thread is being specified. If D=0, then a target thread is being defined; if D=1, then a frame thread is being defined.

METHOD is a 3 bit subfield containing a number between 0 and 3 (D=0) or a number between 0 and 6 (D=1).

If D=0, then METHOD = (0, 1, 2, 3, 4, 5, 6, 7) mod 4, where the 0, ..., 7 indicate methods T0, ..., T7 of specifying a target. Thus, METHOD indicates what kind of Index or Frame Number is required to specify the target, without indicating if the target will be specified in a primary or secondary way. Note that methods 2b, 3, and 7 are not supported by MS-LINK.

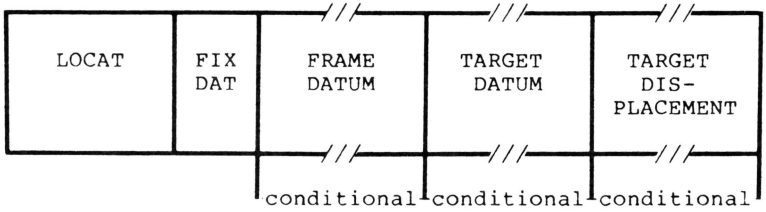
If D=1, then METHOD = 0, 1, 2, 4, 5, corresponding to methods F0, ..., of specifying a frame. Here, METHOD indicates what kind (if any) of Index is required to specify the frame. Note that methods 3 and 5d are not supported by MS-LINK.

THRED is a number between 0 and 3, and associates a Thread Number to the frame or target defined by the THREAD field.

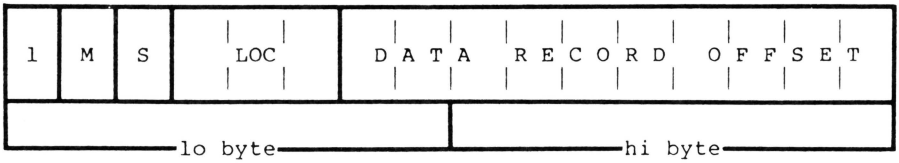
INDEX contains a Segment Index, Group Index, or External Index depending on the specification in the METHOD subfield. This subfield will not be present if F4 or F5 are specified by METHOD.

FIXUP

FIXUP is a field with the following format:



LOCAT is a byte pair with the following format:



M is a one-bit subfield that specifies the mode of the fixups: self-relative (M=0) or segment-relative (M=1).

NOTE

Self-relative fixups may NOT be applied to LIDATA records.

S is a one-bit subfield that specifies that the length of the TARGET DISPLACEMENT subfield, if present (see below), in this FIXUP field will be either two bytes (containing a 16-bit non-negative number, S=0) or three bytes (containing a signed 24-bit number in 2's complement form, S=1).

NOTE

3-byte subfields are a possible future extension, and are not currently supported. Thus, S=0 is currently mandatory.

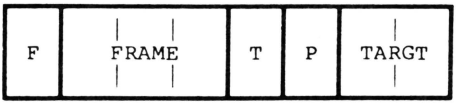
LOC is a 3-bit subfield indicating that the byte(s) in the preceding DATA Record to be fixed up are a 'lobyte' (LOC=0), an 'offset' (LOC=1), a "base" (LOC=2), a "pointer" (LOC=3), or a "hibyte" (LOC=4). (Other values in LOC are invalid.)

The DATA RECORD OFFSET is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of LOCATION (the actual bytes being fixed up) within the preceding DATA record. The DATA RECORD OFFSET is relative to the first byte in the data fields in the DATA RECORDs.

NOTE

If the preceding DATA record is an LIDATA record, it is possible for the value of DATA RECORD OFFSET to designate a "location" within a REPEAT COUNT subfield or a BLOCK COUNT subfield of the ITERATED DATA field. Such a reference is an error. MS-LINK's action on such a malformed record is undefined.

FIX DAT is a byte with the following format:



Note 1: Frame method 2b, F3, and F5d are not supported.

Note 2: Target method T3 and T7 are not supported.

F is a 1-bit subfield that specifies whether the frame for this FIXUP is specified by a thread (F=1) or explicitly (F=0).

FRAME is a number interpreted in one of two ways as indicated by the F bit. If F is zero, FRAME is a number between 0 and 5 and corresponds to methods F0, ..., F5 of specifying a FRAME. If F=1, then FRAME is a thread number (0-3). It specifies the frame most recently defined by a THREAD field that defined a frame thread with the same thread number. (Note that the THREAD field may appear in the same, or in an earlier FIXUPP record.)

T is a 1-bit subfield that specifies whether the target specified for this fixup is defined by reference to a thread (T=1), or is given explicitly in the FIXUP field (T=0).

P is a 1-bit subfield that indicates whether the target is specified in a primary way (requires a TARGET DISPLACEMENT, P=0) or specified in a secondary way (requires no TARGET DISPLACEMENT, P=1). Since a target thread does not have a primary/secondary attribute, the P bit is the only field that specifies the primary/secondary attribute of the target specification.

TARGT is interpreted as a 2-bit subfield. When T=0, it provides a number between 0 and 3, corresponding to methods T0, ..., T3 or T4, ..., T7, depending on the value of P (P can be interpreted as the high-order bit of T0, ..., T7). When the target is specified by a thread (T=1), then TARGT specifies a thread number (0-3).

FRAME DATUM is the "referent" portion of a frame specification, and is a Segment Index, a Group Index, an External Index. The FRAME DATUM subfield is present only when the frame is specified neither by a thread (F=0) nor explicitly by methods F4 or F5 or F6.

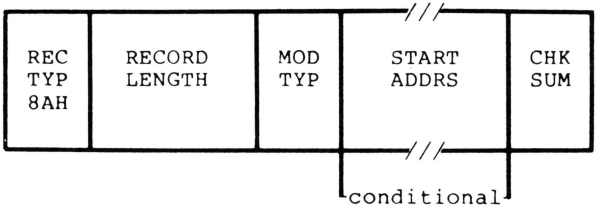
TARGET DATUM is the "referent" portion of a target specification, and is a Segment Index, a Group Index, an External Index or a Frame Number. The TARGET DATUM subfield is present only when the target is not specified by a thread (T=0).

TARGET DISPLACEMENT is the 2-byte displacement required by "primary" ways of specifying TARGETs. This 2-byte subfield is present if P=0.

NOTE

All these methods are described in Section 6.8, "Conceptual Framework for Fixups."

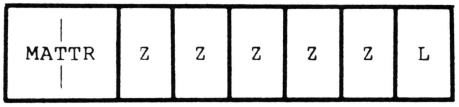
MODULE END RECORD
(MODEND)



This record serves two purposes. It denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true, the execution address is specified.

MOD TYP

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:



MATTR is a 2-bit subfield that specifies the following module attributes:

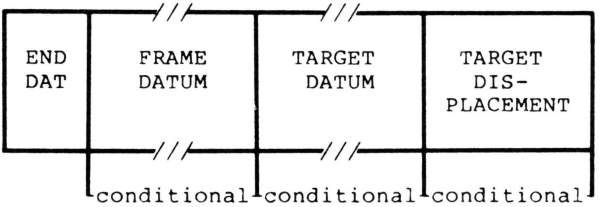
MATTR	MODULE ATTRIBUTE
0	Non-main module with no START ADDRS
1	Non-main module with START ADDRS
2	Main module with no START ADDRS
3	Main module with START ADDRS

L indicates whether the START ADDRS field is interpreted as a logical address that requires fixing up by MS-LINK. (L=1). Note: with MS-LINK, L must always equal 1.

Z indicates that this bit has not currently been assigned a function. These bits are required to be zero.

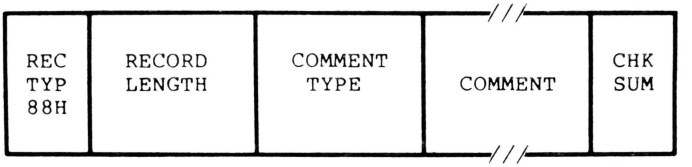
The START ADDRS field (present only if MATTR is 1 or 3) has the following format:

START ADDRS



The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of a logical starting address to a physical starting address is done in exactly the same manner as mapping any other logical address to a physical address as specified in the discussion of fixups and the FIXUPP record. The above subfields of the START ADDRS field have the same semantics as the FIX DAT, FRAME DATUM, TARGET DATUM, and TARGET DISPLACEMENT fields in the FIXUPP record. Only "primary" fixups are allowed. Frame method F4 is not allowed.

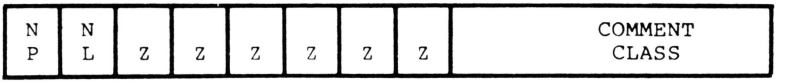
COMMENT RECORD
(COMENT)



This record allows translators to include commentary information in object text.

COMMENT TYPE

This field indicates the type of comment carried by this record. This allows commentary information to be structured for those processes that wish to selectively act on comments. The format of this field is as follows:



The NP (NOPURGE) bit, if 1, indicates that it is not able to be purged by object file utility programs which implement the capability of deleting COMENT record.

The NL (NOLIST) bit, if 1, indicates that the text in the COMMENT field is not to be listed in the listing file of object file utility programs which implement the capability of listing object COMENT records.

The COMMENT CLASS field is defined as follows:

0	Language translator comment.
1	Intel copyright comment. The NP bit must be set.
2-155	Reserved for Intel use. (See note 1.)
156-255	Reserved for users. Intel products will apply no semantics to these values. (See note 2.)

COMMENT

This field provides the commentary information.

Notes:

1. Class value 129 is used to specify a library to add to the Linker's library search list. The comment field will contain the name of the library. Note that unlike all other name specifications, the library name is not prefixed with its length. Its length is determined by the record length. The "NODEFAULTLIBRARYSEARCH" switch causes the linker to ignore all comment records whose class value is 129.

2. Class value 156 is used to specify a DOS level number. When the class value is 156, the comment field will contain a two-byte integer specifying a DOS level number.

6.13 NUMERIC LIST OF RECORD TYPES

*6E RHEADR
*70 REGINT
*72 REDATA
*74 RIDATA
*76 OVLDEF
*78 ENDREC
*7A BLKDEF
*7C BLKEND
*7E DEBSYM
 80 THEADR
*82 LHEADR
*84 PEDATA
*86 PIDATA
 88 COMENT
 8A MODEND
 8C EXTDEF
 8E TYPDEF
 90 PUBDEF
*92 LOCSYM
 94 LINNUM
 96 LNAMES
 98 SEGDEF
 9A GRPDEF
 9C FIXUPP
*9E (none)
 A0 LEDATA
 A2 LIDATA
*A4 LIBHED
*A6 LIBNAM
*A8 LIBLOC
*AA LIBDIC

NOTE

Record types preceded by an asterisk (*) are not supported by the Microsoft Linker. They will be ignored if they are found in an object module.

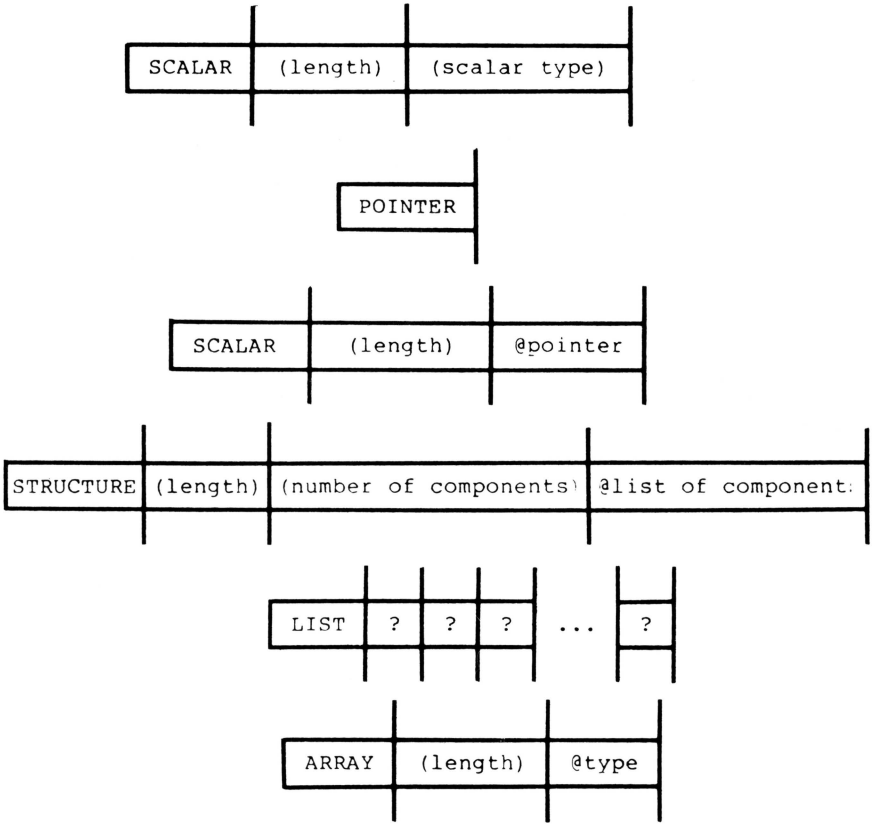
6.14 INTEL TYPE REPRESENTATIONS

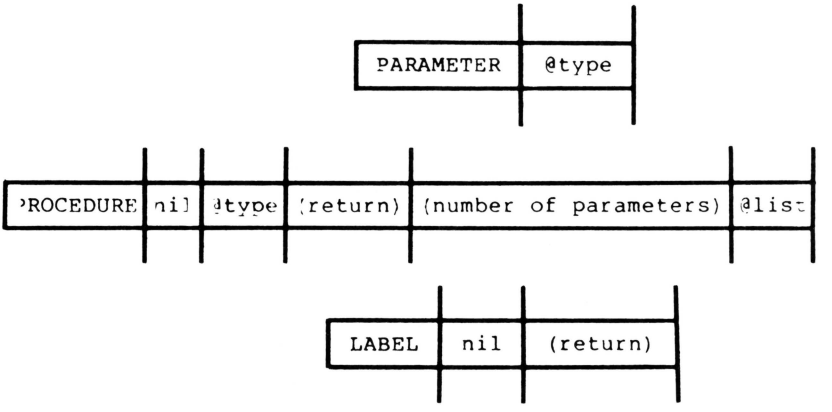
NOTE

The following section describes the Intel standard for type records. Section 6.15, "Microsoft Type Representations," describes the Microsoft standard.

The leaves in the following diagrams may be Numeric Leaves without relations, String Leaves, Index Leaves or Null Leaves. Andleaves and Orleaves are not supported at this time.

Types may be defined by branches of the following forms:





where "(scalar type)" can be either UNSIGNED INTEGER, SIGNED INTEGER, or REAL, "(return)" can be either SHORT or LONG (which indicates, in the case of a LABEL, whether a jump to the label should be a "short" jump or a "long" jump, respectively), and the following values are assigned:

```
 99 INTERRUPT                    114 LONG
100 FILE                         115 SHORT
101 PACKED                       116 PROCEDURE
102 UNPACKED                     117 PARAMETER
103 SET                          118 DIMENSION
104 (Reserved for length)        119 ARRAY
105 CHAMELEON                    120 (Reserved for length)
106 BOOLEAN                      121 STRUCTURE
107 TRUE                         122 POINTER
108 FALSE                        123 SCALAR
109 CHAR                         124 UNSIGNED INTEGER
110 INTEGER                      125 SIGNED INTEGER
111 CONST                        126 REAL
112 (Reserved for length)        127 LIST
113 LABEL
```

NOTE

The previous (decimal) values are chosen for the convenience of utility programs. All numbers are different (although conceptually there is no reason why REAL and SCALAR, for example, can't be the same number), and are rather large, so that object module display programs may correctly decide whether to represent a Numeric Leaf as a number or as an identifier, make this choice correctly most of the time, and never give a wrong identifier.

6.15 MICROSOFT TYPE REPRESENTATIONS FOR COMMUNAL VARIABLES

This section defines the Microsoft standard for communal variable allocation on the 8086 and 80286.

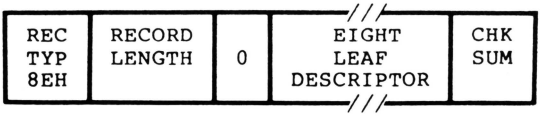
A communal variable is an uninitialized public variable whose final size and location are not fixed at compile time. Communal variables are similar to FORTRAN common blocks in that if a communal variable is declared in more than one object module being linked together, then its actual size will be the largest size specified in the several declarations. In the C language, all uninitialized public variables are communal. The following example shows three different declarations of the same C communal variable:

```
char    foo[4];         /* In file a.c */
char    foo[1];         /* In file b.c */
char    foo[1024];      /* In file c.c */
```

If the objects produced from a.c, b.c, and c.c are linked together, then the linker will allocate 1024 bytes for the char array foo.

A communal variable is defined in the object text by an external definition record (EXTDEF) and the type definition record (TYPDEF) to which it refers.

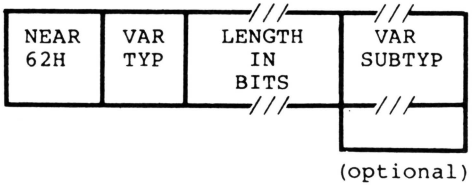
The TYPDEF for a communal variable has the following format:



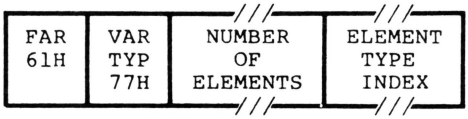
The EIGHT LEAF DESCRIPTOR field has the following format:

The EN field specifies whether the next 8 leaves in the LEAF DESCRIPTOR field are EASY (bit = 0) or NICE (bit = 1). This byte is always zero for TYPDEFS for communal variables.

The LEAF DESCRIPTOR field has one of the following two formats. The format for communal variables in the default data segment (near variables) is as follows:



The VARiable TYPe field may be either SCALAR (7BH), STRUCT (79H), or ARRAY (77H). The VAR SUBTYP field (if any) is ignored by the Linker. The format for communal variables not in the default data segment (far variables) is as follows:



The VARiable TYPe field must be ARRAY (77H). The length field specifies the NUMBER OF ELEMENTS, and the ELEMENT TYPE INDEX is an index to a previously defined TYPDEF whose format is that of a near communal variable.

The format for the LENGTH IN BITS or NUMBER OF ELEMENTS fields is the same as the format for the LEAF DESCRIPTOR field, described in the TYPDEF record format section of this manual.

Link time semantics:

All EXTDEFs referencing a TYPDEF of one of the previously described formats are treated as communal variables. All others are treated as externally defined symbols for which a matching public symbol definition (PUBDEF) is expected. A PUBDEF matching a communal variable definition will override the communal variable definition. Two or more matching communal variable definitions must agree on whether the variable in question is near or far; the Linker issues a warning if there is a conflict.

If the variable is near, then its size is the largest of the sizes specified for it. If the variable is far, then the Linker issues a warning if there are conflicting array element size specifications; if there are no such conflicts, then the variable's size is the element size times the largest number of elements specified. The sum of the sizes of all near variables must not exceed 64K bytes. The sum of the sizes of all far variables must not exceed the size of the machine's addressable memory space.

"Huge" communal variables:

A far communal variable whose size is larger than 64K bytes will reside in segments that are contiguous (8086) or have consecutive selectors (80286). No other data items will reside in the segments occupied by a huge communal variable.

Allocate Memory (Function 48H) 1-147
Archive bit 3-7
ASCIZ 1-155
ASCIZ string 1-160
Autoexec file 3-1

Backward compatibility 1-7
BASE 6-9
Block device
  example 2-23

Canonic 6-4
Carry flag 1-6
Case-mapping call 1-113
Change Current Directory (Function 3BH) 1-119
Change Mode (Function 43H) . . 1-135
Character device
  example 2-37
Class name 6-5
Close File Handle (Function 3EH) 1-125
Cluster 3-4
Combination Attribute 6-23
COMENT 6-48
Command processor 3-1
Command.com 3-1
COMMENT RECORD 6-48
Complete name 6-5
Comspec= 4-3
Control blocks 4-1
Control information 5-1
CONTROL-C Check (Function 33H) 1-106
CONTROL-C Exit Address (Interrupt 23H) 1-26
Create Directory (Function 39H) 1-115
Create Handle (Function 3CH) . 1-121

Delete Entry (Function 41H) . 1-131
Device drivers 3-9
  creating 2-6
  dumb 2-13
  example 2-23, 2-37
  installing 2-7
  smart 2-13
Device handles 1-9
Directory entry 1-10
  fields 1-10
  format 1-10
Disk allocation 3-3

Disk directory 3-4
Disk errors, Interrupt 20H . . 1-29
Disk formats
  IBM 3-11
  standard MS-DOS 3-11
Disk transfer address 4-3
DTA 1-160
Duplicate a File Handle (Function 45H) 1-141

EIGHT LEAF DESCRIPTOR 6-28
End Process (Function 4CH) . . 1-157
Error checking 1-5
Error codes, Interrupt 20H . . 1-27
Error handling 1-14, 3-1
EXE files 5-1
Execute Program (Function 4BH) 1-153
EXTDEF 6-33
Extended FCB 1-17
EXTERNAL NAMES DEFINITION RECORD 6-33

FAT 3-9
FAT ID byte 2-18
Fatal Error Abort Address (Interrupt 24H) 1-26
FCB 1-15
Fields 1-15
File allocation table 3-9
File attributes 1-11
File control block
  definition 1-15
  extended 1-17
  fields 1-15
File Control Block (FCB) . . . 1-15
File-related function requests 1-8
Find First File (Function 4EH) 1-160
Find Next File (Function 4FH) 1-163
FIXUP RECORD 6-40
FIXUPP 6-40
Fixups 6-8
  segment-relative 6-10, 6-15
  self-relative 6-10, 6-14
Force Duplicate File handle (Function 46H) 1-143
Format 3-4
FRAME
  DEFINITION 6-3
  SPECIFYING 6-12
FRAME NUMBER 6-3
Free Allocated Memory (Function 49H) 1-149
Function Request 4CH 1-7
Function requests 1-1, 1-36
  calling 1-4
  definition 1-2
  file-related 1-8
  Function 00H 1-37
  Function 01H 1-39

Function 02H 1-40
Function 03H 1-41
Function 04H 1-42
Function 05H 1-43
Function 06H 1-45
Function 07H 1-47
Function 08H 1-48
Function 09H 1-49
Function 0BH 1-52
Function 0CH 1-53
Function 0DH 1-54, 1-69
Function 0EH 1-55
Function 0FH 1-57, 1-71
Function 10H 1-59
Function 11H 1-61
Function 12H 1-63
Function 13H 1-65
Function 14H 1-67
Function 15H 1-69
Function 16H 1-71
Function 17H 1-73
Function 19H 1-75
Function 1AH 1-76
Function 21H 1-78
Function 22H 1-80
Function 23H 1-82
Function 24H 1-84
Function 25H 1-26, 1-85
Function 27H 1-87
Function 28H 1-89
Function 29H 1-91
Function 2AH 1-94
Function 2BH 1-96
Function 2CH 1-98
Function 2DH 1-99
Function 2EH 1-101
Function 2FH 1-103
Function 30H 1-104
Function 31H 1-105
Function 33H 1-106
Function 35H 1-108
Function 36H 1-110
Function 38H 1-112
Function 39H 1-115
Function 3AH 1-117
Function 3BH 1-119
Function 3CH 1-121
Function 3DH 1-123
Function 3EH 1-125
Function 3FH 1-127
Function 40H 1-129
Function 41H 1-131
Function 42H 1-133
Function 43H 1-135
Function 44H 1-137

Function 45H 1-141
Function 47H 1-145
Function 48H 1-147
Function 49H 1-149
Function 4AH 1-151
Function 4BH 1-153
Function 4CH 1-157
Function 4DH 1-159
Function 4EH 1-160
Function 4FH 1-163
Function 54H 1-165
Function 56H 1-166
Function 57H 1-168
Function 0AH 1-50
Function Requests (Function 46H) 1-143

Get Country Data (Function 38H) 1-112
Get Current Directory (Function 47H) 1-145
Get Disk Free Space (Function 36H) 1-110
Get DOS Version Number (Function 30H) 1-104
Get DTA (Function 2FH) 1-103
Get Vector (Function 35H) . . 1-108
Get/Set Date or Time (Function 57H) 1-168
Get/Verify State (Function 54H) 1-165
Group 6-4
GROUP DEFINITION RECORD . . . 6-26
GRPDEF 6-26

Handles 1-9
definition 1-9
device 1-9
Header 5-1
HIBYTE 6-9
Hidden file 3-6
High-level language 1-5

I/O Control for Devices (Function 44H) 1-137
IBM disk format 3-11
Index fields 6-7
Indices 6-7
Instruction pointer 4-5
Intel record types 6-51
Internal stack 1-5
Interrupt 21H 1-5
Interrupt handler . . . . . 1-1 to 1-2
Interrupt handlers 4-1
Interrupt routine addresses . 1-2
Interrupts
definition 1-1
Interrupt 20H 1-23, 1-27, 1-29, 1-37
Interrupt 21H 1-25
Interrupt 22H 1-26
Interrupt 23H . . . . . 1-26, 1-39 to 1-40, 1-43, 1-50
Interrupt 24H . . . . . 1-26 to 1-27

Interrupt 25H 1-31
Interrupt 26H 1-33
Interrupt 27H 1-35
Interrupts, issuing 1-4
IO.SYS 3-7

LEDATA 6-36
LIDATA 6-37
LINE NUMBERS RECORD 6-35
LINNUM 6-35
LIST OF NAMES RECORD 6-21
LNAMES 6-21
Load module 5-1, 5-3
LOBYTE 6-9
LOCATION
TYPES 6-9
LOGICAL ENUMERATED DATA RECORD 6-36
LOGICAL ITERATED DATA RECORD . 6-37
Logical sector 3-9
Logical sector numbers 3-10
Logical Segment 6-3
LSEG 6-3

Macro definitions 1-170
MAS 6-2
Memory Address Space 6-2
Microsoft record types 6-55
MODE 6-10
MODEND 6-46
MODULE 6-2
MODULE END RECORD 6-46
Module header record 6-6
Move Pointer (Function 42H) . 1-133
MS-DOS 2.0 User's Guide . . . 1-8
MS-DOS initialization 3-1
MS-DOS memory map 4-1
MS-LINK 5-1
MSDOS.SYS 3-1, 3-7

Numeric record types 6-50

Object Module Formats 6-2
Object module record formats . 6-2
OFFSET 6-9
OMF 6-2
Open Handle (Function 3DH) . . 1-123
Overlay name 6-5

PARAGRAPH Number 6-4
Parameter block 1-154
Physical Segment 6-3
Program segment 4-2
Program segment prefix 1-4, 1-15, 1-154, 4-2, 5-3
Program Terminate (Interrupt 20H) 1-23

PSEG 6-3
PSEG Number 6-4
PUBDEF 6-30
PUBLIC NAMES DEFINITION RECORD 6-30

Read File or Device (Function 3FH) 1-127
Read only memory 3-1
Record format
  sample 6-18
Record formats 6-1
Record order 6-16
Record types
  Intel 6-51
  Microsoft 6-55
  numeric 6-50
Registers 1-5
  treatment 1-5
Relocatable memory images . . 6-1
Relocation information 5-1
Relocation item offset value . 5-3
Relocation table 5-2
Remove Directory (Function 3AH) 1-117
Rename a File (Function 56H) . 1-166
Returning control to MS-DOS . 1-7
ROM 3-1
Root directory 3-4

SEGDEF 6-22
Segment addressing 6-6
Segment definition 6-6
SEGMENT DEFINITION RECORD . . 6-22
Segment name 6-5
Segment-relative fixups . . . 6-10, 6-15
Self-relative fixups 6-10, 6-14
Set Block (Function 4AH) . . . 1-151, 4-5
Start segment value 5-3
Symbol definition 6-7
System calls
  definition 1-1
  types 1-1
System file 3-7
System prompt 3-2

T-MODULE 6-2
T-module Header Record (THEADR) 6-20
TARGET 6-10
Terminate Address (Function 4CH) 4-2
Terminate Address (Interrupt 22H) 1-26
Terminate But Stay Resident (Function 31H) 1-105
THEADR 6-20
TYPDEF 6-27
TYPE DEFINITION RECORD 6-27

User stack 4-1

Volume label 3-7

Wait (Function 4DH) 1-159
Write File or Device (Function 40H) 1-129

XENIX
  compatible calls 1-3