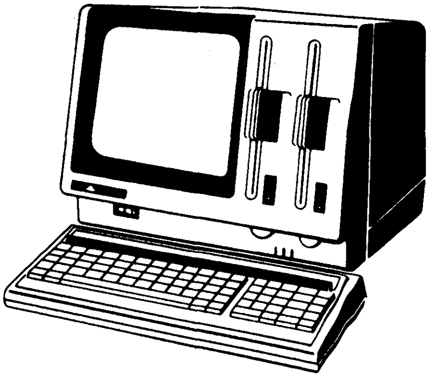
Advanced
Personal Computer



MS™-DOS System Reference Guide

NEC

NEC Information Systems, Inc.

819-000103-2001 REV 01
9-83

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.

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.
Dept: Publications
1414 Mass. Ave.
Boxborough, MA 01719

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.

819-000102-8D01

APC Advanced Personal Computer™

NEC
NEC Information Systems, Inc.

Program Name (as it appears on diskette label)

____

Serial Number ____________________ Date Purchased ____________

Dealer Name and City ______________________________________

Your Name ____

Your Address ____

City ____________________ State ________ ZIP __________

"Warranty Requires Return of This Card"

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 386 LEXINGTON MA

POSTAGE WILL BE PAID BY ADDRESSEE

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

Contents

Page

Chapter 1 MS-DOS System Overview

Chapter 2 MS-DOS System Calls

Contents (cont'd)

Page

Contents (cont'd)

Page

Contents (cont'd)

Page

Contents (cont'd)

Page

Contents (cont'd)

Page

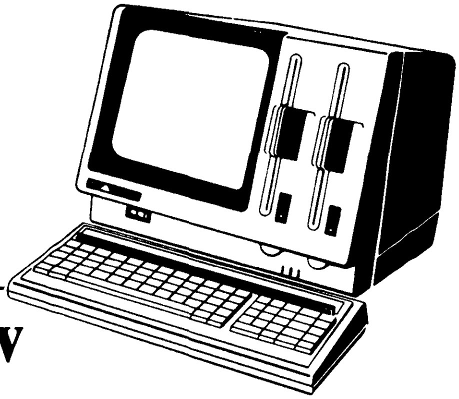
Appendix The MS-DOS Interrupt Vectors

Tables

Illustrations

Chapter 1

MS-DOS System Overview



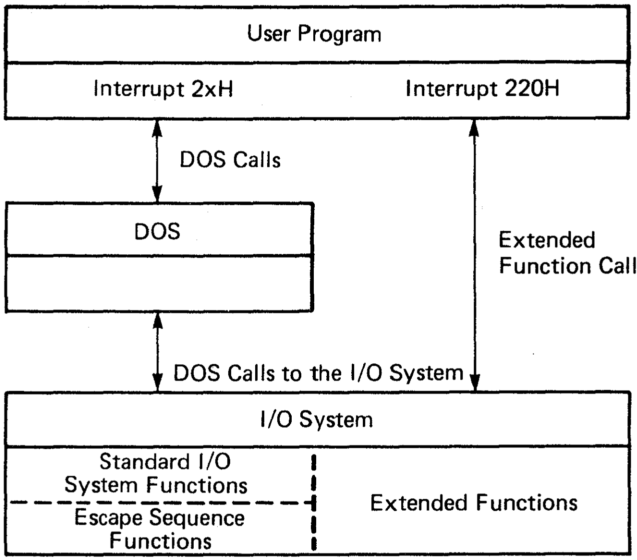
The MS™-DOS operating system for the APC is divided into two subsystems: the Disk Operating System (DOS) and the I/O System. The DOS routines are for file management, data blocking and deblocking, and a variety of internal functions. I/O System routines include standard functions, extended functions, and escape sequence functions. Standard I/O routines perform basic functions, such as program termination and absolute disk reads or writes. The extended I/O routines add facilities like music playing and direct CRT I/O. APC escape sequence functions are called by user programs to control screen I/O.

MS-DOS SOFTWARE

The MS-DOS software consists of three programs: MSDOS.SYS, IO.SYS, and COMMAND.COM.

- MSDOS.SYS provides access to DOS routines. When these routines are called by a user program, they accept high-level information through register and control block contents. Then for device operations, they translate the requirement into one or more calls to IO.SYS (see below) to complete the request. Thus, MSDOS.SYS calls both DOS and standard functions for the I/O System.
- IO.SYS executes all the hardware dependent routines for the APC. In addition to the standard I/O System functions called by MSDOS.SYS, this program executes the extended I/O System functions and the APC escape sequence functions. When user programs issue calls for extended I/O functions, they access IO.SYS directly, bypassing MSDOS.SYS. IO.SYS receives requests to perform escape sequence functions through MSDOS.SYS, as it does for standard I/O functions.
- COMMAND.COM (Command Processor) interprets the MS-DOS commands entered at the APC keyboard, converting them into calls to MSDOS-.SYS. How the Command Processor resides in memory and details on its operations are given in the section THE COMMAND PROCESSOR.

The following illustration represents the interactions of user programs and the MS-DOS subsystems.



The user program issues any calls to the DOS through Interrupts 20H to 27H. (For a further explanation of these interrupts, see Chapter 2.) To use an extended function, the program must issue a call through Interrupt 220H. (Chapter 3 presents the extended I/O functions for the I/O System.)

MS-DOS INITIALIZATION

MS-DOS initialization consists of several steps. First, a ROM (Read Only Memory) bootstrap obtains control and reads the boot sector off the MS-DOS system diskette. The loaded bootstrap then loads IO.SYS. Next, IO.SYS loads MSDOS.SYS. Finally, MSDOS.SYS loads COMMAND.COM.

Figure 1-1 illustrates both the logical memory structure of the APC and a memory map. MS-DOS occupies memory beginning after the interrupt vectors at absolute address 400H. The interrupt vectors for the APC are categorized as the CPU, device, MS-DOS reserved, user, and APC-reserved interrupt vectors. See Appendix A for a description of these vectors.

A resident portion of IO.SYS, remaining in memory after its loading tasks, follows the interrupt vectors.

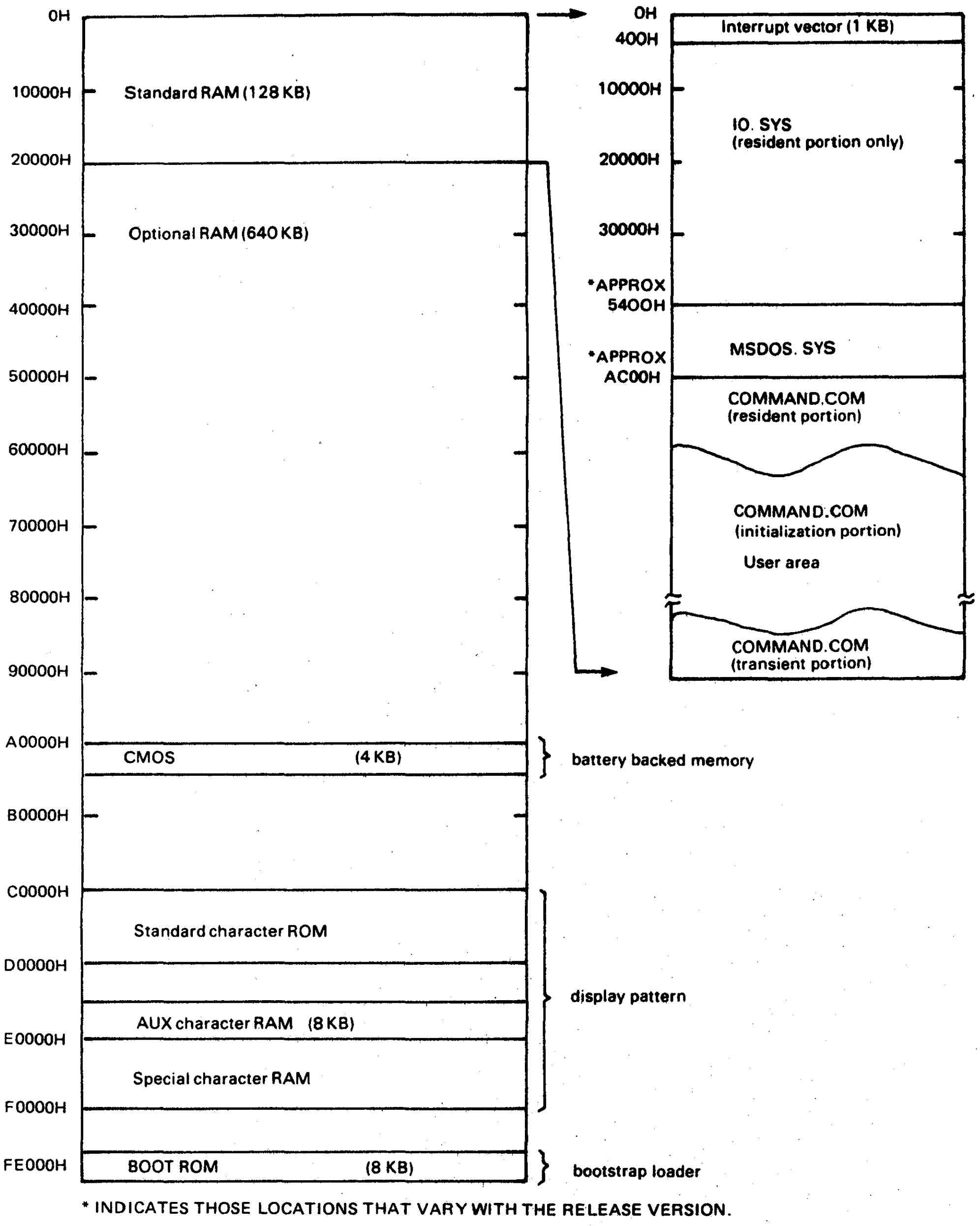


Figure 1-1 Logical Memory Structure of the APC and Memory Map.

MSDOS.SYS resides in memory after the resident portion of IO.SYS.

Last, COMMAND.COM occupies memory after AC00H (approximately). The Command Processor code is divided into three sections:

- A resident portion that resides in memory immediately following MSDOS-.SYS and its data area. This portion contains routines to process interrupt types 22H (Terminate Address), 23H (CTRL-C Exit Address), and 24H (Fatal Error Abort Address), as well as a routine to reload the transient portion of the Command Processor (see item 3), if needed. Note that all standard MS-DOS error handling is done within this portion of COMMAND.COM. This includes displaying error messages and interpreting the replies to the messages displayed with "Abort, Retry, or Ignore."
- An initialization portion that follows the resident portion (actually in the user area) and is given control during startup. This section contains the AUTOEXEC.BAT file processor setup routine. The initialization portion determines the segment address at which programs can be loaded. It is overlaid by the first program COMMAND.COM loads because it is no longer needed.
- A transient portion that is loaded at the high end of memory. This portion contains all of the internal command processors and the batch file processor.

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

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

ALLOCATION OF DISK SPACE FOR FILES

MS-DOS organizes the space on disk ("disk" will be used from this point on to refer to both diskette and hard disk, unless otherwise stated) as follows:

- reserved area -variable size
- first copy of the File Allocation Table - variable size
- second copy of the File Allocation Table - variable size (optional)
- root directory - variable size
- data area.

Space for a file is allocated in the data area only when needed; it is not pre-allocated. The space is allocated one cluster (unit of allocation) at a time. A cluster is always one or more consecutive sectors, and all of the clusters for a file are "chained" together in the File Allocation Table (FAT), containing pointers to the individual files on the disk. There is usually a second FAT kept, which is a copy of the first, for consistency of format. Should the disk develop a bad sector in the middle of the first FAT, the second can be used. This avoids data loss due to a defective disk.

Cluster Arrangement

Clusters are arranged on disk to minimize head movement on multi-sided media. All of the space on a track (or cylinder) is allocated before the next track is selected. Consecutive sectors on the lowest-numbered head are used, followed by 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.

File Allocation Table Format

The File Allocation Table consists of 12-bit entries (1.5 bytes) for each cluster on the disk. The first two FAT entries (24 bits) map a portion of the directory. These FAT entries contain indicators of the size and format of the disk. The first byte of the two entries designates the type of disk: single- or double-sided and single- or double-density. The second and third bytes always contain FFFH.

The third FAT entry begins the mapping of the data area (cluster 002). Files in the data area are not necessarily written sequentially on the disk. The data area space is allocated one cluster at a time; clusters already allocated are skipped. The first free cluster found will be the next cluster allocated, regardless of its physical location. This permits the most efficient use of disk space because clusters made available by erasing files can be allocated for new files. (Refer to the description of the MS-DOS 2.0 File Allocation Table format in the *MS-DOS System Programmer's Guide* for more information.)

MS-DOS ROOT DIRECTORY STRUCTURE

The MS-DOS FORMAT utility (invoked by the HDFORMAT external command) initially builds the root directory for all diskettes. This utility allocates the root directories for hard disk volumes. The location (logical sector number) and the maximum number of entries for a root directory can be obtained through device driver interfaces.

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

All directory entries are 32 bytes in length. Table 1-1 lists the fields in an entry, giving their names, sizes, and byte offsets in hexadecimal and decimal.

Table 1-1 Directory Entry Fields

NAME	SIZE (BYTES)	OFFSET	
		HEX	DECIMAL
Filename	8	00H-07H	0-7
File extension	3	08H-0AH	8-10
File attributes	1	0BH	11
Reserved	10	0CH-15H	12-21
Time of last write	2	16H,17H	22,23
Date of last read	2	18H,19H	24,25
Reserved	2	1AH,1BH	26,27
File size	4	1CH-1FH	28,31

The following provides more information on the directory entry fields.

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

00H	Has never been used. This is used to limit the length of directory searches for performance reasons.
E5H	Was used, but the file has been erased.
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 root directory).

  Any other character is the first character of a filename.

- Filename extension (offset 08H). Three characters, left-aligned and padded (if necessary) with blanks. This field can be all blanks (no extension).
- File attributes (offset 0BH).

The following are the values of the attributes.

hex value	binary value	
01H	0000 0001	File is marked read-only. An attempt to open the file for writing using function call 4DH results in an error code being returned. This value can be used with values below.
02H	0000 0010	Hidden file. The file is excluded from normal directory searches.
04H	0000 0100	System file. The file is excluded from normal directory searches.
07H	0000 0111	Changeable with CHGMOD.
08H	0000 1000	The entry contains the volume label in the first 11 bytes. The entry contains no other usable information (except the date and time of volume creation) and may exist only in the root directory.
0AH	0001 0000	The entry defines a sub-directory, and is excluded from normal directory searches. Note that a directory listing gives only the highest-level directory name where there are parent directories involved.
16H	0001 0110	Hard attributes for FINDENTRY.
20H	0010 0000	Archive bit. The bit is set to on whenever the file has been written to and closed. This bit can be used along with other attribute bits. Note that 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 Function 43H.

- Reserved (offset 0CH). Reserved for MS-DOS.
- Time of Last Write (offset 16H). The time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:

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

Or, described as mapped bits:

```
        hh               m m               x x
   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

where:

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

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

```
Offset 19H
   Y  Y  Y  Y  Y  Y  Y  M
   15                 98
Offset 18H
   M  M  M  D  D  D  D  D
          54              0
```

Or viewed as mapped bits:

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

where:

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

- Reserved (offset 1AH). Starting cluster or the relative 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. (Refer to information on the File Allocation Table in the *MS-DOS System Programmer's Guide* for details on converting cluster numbers to logical sector numbers.)
- 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.

PERIPHERAL DEVICE AND DEVICE CONTROLLER CHARACTERISTICS

The following are characteristics of the peripheral devices attached to the APC and their device controllers.

The APC Screen

The APC screen is controlled by the Graphic Display Controller (GDC). GDC

- generates the basic video raster timing
- partitions the screen into areas for independent scrolling
- performs zooming and panning operations
- modifies video-display memory and moves data
- calculates the video-display memory address
- performs DMA operations between the main memory and video-display memory.

Some further characteristics of the CRT-control design are

- a display buffer independent of system memory
- an 80-character by 25-line screen (2000 characters)
- a direct drive output
- an 8-dot by 19-dot character box
- a 7-dot by 11-dot character box
- 16-dot by 16-dot special programmable characters.

Printers

The printer driver supplied with MS-DOS controls the following NEC printers:

- the NEC 8023 Dot Matrix Printer operating at 100 characters/second with 136 characters/line
- the NEC Spinwriter 3530 operating at 350 words/minute.

Diskette and Hard Disk Drives

MS-DOS provides drivers for controlling four diskette drives and two hard disk drives.

DISKETTE ATTRIBUTES

The APC uses 8-inch (200 mm) diskettes for storing information. In the APC system, "diskette" is the term used for "floppy disk," "floppy," or "disk."

You can use two types of diskette on the APC. One is a single-sided, single-density diskette (called FD1); the other is a double-sided, double-density diskette (called FD2D).

The FD1 diskette uses the IBM 3740 format with the following characteristics:

- 128 bytes per sector, soft sectored
- 4 sectors per allocation unit
- 1 reserved sector
- 2 FATs
- 68 directory entries in the root directory area
- 77 x 26 sectors.

The MS-DOS FORMAT command does not format an FD1 diskete for use as a system diskette. The Boot Loader that resides in ROM will not load the MS-DOS programs from an FD1 diskette, so placing these programs on this type of diskette is of no use.

The FD2D diskette has the IBM-compatible format of

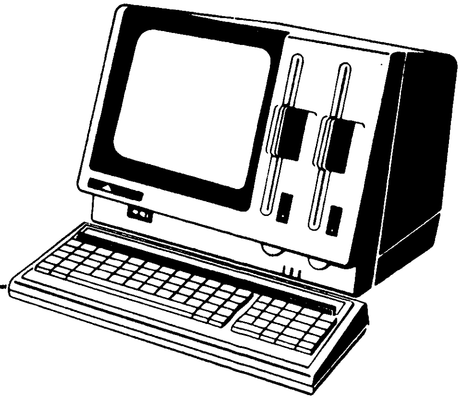
- 1024 bytes per sector, soft sectored
- 1 sector per allocation unit
- 1 reserved sector
- 2 FATs
- 192 directory entries in the root directory area
- 77 x 8 x 2 sectors.

The FORMAT command formats an FD2D diskette for use as a system diskette or as a data diskette. However, a system diskette will not have a standard Microsoft boot sector format.

HARD DISK ATTRIBUTES

Hard disk configuration attributes are

- 512 bytes per sector
- 2 sectors per allocation unit
- 0 reserved sectors
- 2 FATs
- 1024 entries in a root directory area
- a variable number of sectors, as specified by the user during Hard Disk Formatter (HDFORMAT) execution.



Chapter 2

MS-DOS System Calls

MS-DOS uses two types of system calls: interrupts and function requests.

Interrupts are the lowest-level primitives available in the operating system. They provide access to standard function routines in the I/O System.

Function requests provide access to primitive routines in the DOS. The DOS primitives, in turn, call the interrupts to perform their processing.

PROGRAMMING CONSIDERATIONS

System calls free you from having to invent your own ways to perform primitive functions. They make it easier to write machine-independent programs. Some knowledge of system control blocks is required to use the disk input/output system calls. These control blocks are described in this chapter.

Calling from the MACRO-86 Macro Assembler™

System calls can be invoked from the MACRO-86 Macro Assembler simply by moving any required data into registers and issuing an interrupt. Some of the calls destroy registers, so you may have to save registers before using a system call. The system calls can be used in macros and procedures to make your programs more readable.

Calling from a High-Level Language

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

Returning Control to MS-DOS

Control can be returned to MS-DOS in three ways:

- Call interrupt 20H:

  ```
  INT 20H
  ```

  This is the quickest way.

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

  ```
  JMP 0
  ```

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

- Call Function Request 00H:

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

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

Console and Printer Input/Output Calls

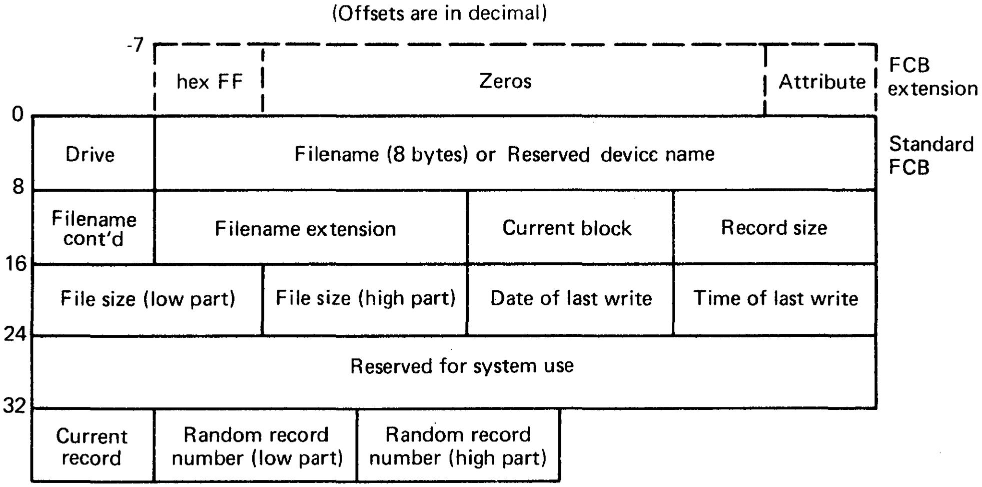
The system calls for the console (keyboard) and printer let you read from and write to the console device and print on the printer without using any machine-specific codes. You can still take advantage of specific capabilities (display attributes such as positioning the cursor or erasing the screen,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

Disk I/O System Calls

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

FILE CONTROL BLOCK FORMAT

The Program Segment Prefix control block, built by MS-DOS for each program to be executed, includes room for two File Control Blocks (FCBs) at offsets 5CH and 6CH. The system call descriptions refer to unopened and opened FCBs. An unopened FCB is one that contains only a drive specifier and filename, which can contain wild card characters (* and ?). An opened FCB contains all fields filled by the Open File system call (Function 0FH). Figure 2-1 illustrates the format of the FCB.



Unshaded areas must be filled in by the using program.
Shaded areas are filled in by the DOS and must not be modified.

Figure 2-1 File Control Block

File Control Block Fields

Table 2-1 lists each field of the FCB, giving its size and offset in decimal and hexadecimal

Table 2-1 File Control Block Fields

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

Additional information about the FCB fields is as follows:

- Drive Number (offset 00H). Specifies the drive; 1 means drive A, 2 means drive B, and so forth. If the FCB is to be used to create or open a file, this field can be set to 0 to specify the default drive; the Open File system call Function (0FH) sets the field to the number of the default drive.
- Filename (offset 01H). Eight characters, left-aligned and padded (if necessary) with blanks. If you specify a reserved device name (such as LPT1), do not put a colon at the end.
- Extension (offset 09H). Three characters, left-aligned and padded (if necessary) with blanks. This field can be all blanks (no extension).
- Current Block (offset 0CH). Points to the block (group of 128 records) that contains the current record. This field and the Current Record field (offset 20H) make up the record pointer. This field is set to 0 by the Open File system call.
- Record Size (offset 0EH). The size of a logical record in bytes. Set to 128 by the Open File system call. If the record size is not 128 bytes, you must set this field after opening the file.
- File Size (offset 10H). The size of the file in bytes. The first word of this 4-byte field is the low-order part of the size.
- Date of Last Write (offset 14H). The date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

```
Offset 15H
    Y   Y   Y   Y   Y   Y   Y   M
   15                       9   8

Offset 14H
    M   M   M   D   D   D   D   D
              5 4                0
```

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

```
Offset 17H
    H   H   H   H   H   M   M   M
   15               11  10

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

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

- Current Record (offset 20H). Points to one of the 128 records in the current block. This field and the Current Block field (offset 0CH) make up the record pointer. This field is not initialized by the Open File system call. You must set it before doing a sequential read or write to the file.
- Relative Record (offset 21H). Points to the currently selected record, counting from the beginning of the file (starting with 0). This field is not initialized by the Open File system call. You must set it before doing a random read or write to the file. If the record size is less than 64 bytes, both words of this field are used. If the record size is 64 bytes or more, only the first three bytes are used.

NOTE

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

Extended File Control Block

The Extended File Control Block (Extended FCB) is used to create or search for directory entries of files with special attributes. It adds the following seven-byte prefix consisting of a name, size, and decimal offset to the normal FCB:

Byte	Function
FCB-7	Flag byte containing FFH to indicate an Extended FCB.
FCB-6 to FCB-2	Reserved.
FCB-8	Attribute byte (02H = hidden file; 04H = system file). Also refer to Function Request 11H (Search for First Entry) for details on using the attribute bits during directory searches. This function allows applications to define their own files as hidden and thereby exclude them from directory searches. It also allows for selective directory searches.

Any references in the MS-DOS function calls to an FCB, whether opened or unopened, may designate either a normal or extended FCB. If using an extended FCB, you should set the appropriate register to the first byte of the prefix rather than the drive-number field.

SYSTEM CALL DESCRIPTIONS

The system calls to DOS and standard I/O System routines are described in the pages that follow. The descriptions of the system calls provide some or all of the following information:

- A representation of the registers that shows their contents before and after the system call. Many system calls require that parameters be loaded into one or more registers before the call is issued. Most calls return information in the registers (usually a code that indicates the success or failure of the operation).
- More information about the register contents required before the system call.
- An explanation of the processing performed.
- Error returns from the system call, if any.
- An example of its use.

  A macro is defined for each system call, then used in an example. In addition, a few other macros are included in the examples. These macros make the examples appear more like complete programs, rather than isolated uses of the system calls. All macro definitions are listed at the end of the chapter.

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

In their detailed descriptions, system calls are listed in numeric order. The interrupts are described first, then the function requests.

NOTE

> Unless otherwise stated, all numbers in the system call descriptions — both text and code — are in hex.

Interrupts

MS-DOS reserves interrupts 20H through 3FH for its own use. The table of interrupt routine addresses (vectors) is maintained in locations 80H-FCH. Table 2-2 lists the interrupts in numeric order.

Table 2-2 MS-DOS Interrupts

INTERRUPT		DESCRIPTION
HEX	DEC	
20H	32	Program Terminate
21H	33	Function Request
22H	34	Terminate Address
23H	35	CTRL-C Exit Address
24H	36	Fatal Error Abort Address
25H	37	Absolute Disk Read
26H	38	Absolute Disk Write
27H	39	Terminate But Stay Resident
28H-40H	40-64	RESERVED — DO NOT USE

User programs should issue only interrupts 20H, 21H, 25H, 26H, and 27H.

NOTE

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

PROGRAM TERMINATE

ENTRY →

CS Segment address
of Program
Segment Prefix

INTERRUPT 20H

RETURN →

Interrupt 20H terminates the current process and returns control to its parent process. All open file handles are closed and the disk cache is cleaned. This interrupt is almost always used in .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 indicated offsets of the Program Segment Prefix.

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

All file buffers are flushed to disk.

CAUTION

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 Function 10H for a description of the Close File system call.

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

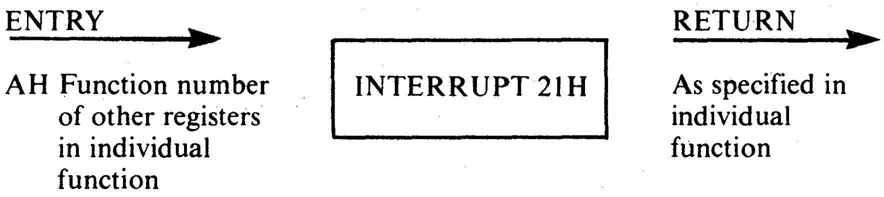
Macro Definition:

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

Example:

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

FUNCTION REQUEST



ENTRY

AH Function number
of other registers
in individual
function

INTERRUPT 21H

RETURN

As specified in
individual
function

Interrupt 21H allows for calling of a specified function. The AH register must contain the number of the system function. See the section FUNCTION REQUESTS for a description of the MS-DOS system functions.

NOTE

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

Example:

To call the Get Time function:

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

INTERRUPTS 22H, 23H, AND 24H

Interrupts 22H, 23H, and 24H are not true interrupts, but 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. If a program executes a second program, it must set the terminate address before it creates the segment for the second program. Otherwise, when the second program terminates, it will transfer to the first program's termination address.

Interrupt 23H — CTRL-C Exit Address

If you press CTRL-C during keyboard input or display output, control transfers to the address at offset 0EH of the Program Segment Prefix. This address is copied into the Program Segment Prefix, from the Interrupt 23H vector, when the segment is created.

If the CTRL-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 CTRL-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 CTRL-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 CTRL-C address, termination of the second program restores the CTRL-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 address at offset 12H of the Program Segment Prefix. This address is copied into the Program Segment Prefix, from the Interrupt 24H vector, when the segment is created.

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. This routine retries the disk operation, then gives the user the choice of aborting, retrying the operation, or ignoring the error.

The following sections provide information for interpreting the error codes, managing the registers and stack, and controlling the system's response to an 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 a bad memory image of the File Allocation Table. No further information is available.

If bit 7 of AH is 0, it is a disk error; the following registers describe the failure.

AL identifies the drive (0 = A, 1 = B, and so on.).

AH identifies the operation and affected area.

The lower half of DI identifies the error.

Table 2-3 describes the operation code in AH.

Table 2-3 Disk Error Operation Codes (AH)

CODE	OPERATION	AFFECTED AREA
0	Read	System files
1	Write	
2	Read	File Allocation Table
3	Write	
4	Read	
5	Write Directory	
6	Read	Data area
7	Write	

Table 2-4 describes the error code in the lower half of DI.

Table 2-4 Disk Error Codes (Lower Half of DI)

CODE	MEANING
0	Attempt to write on write-protected diskette
2	Drive not ready
4	Data error
6	Seek error
8	Sector not found
0AH	Write fault
0CH	General disk failure

RETRIES

The DS, BX, CX, and DX registers contain the required data for a retry of the operation. Specify the action to be taken by putting one of the following values in AL and executing an IRET.

Value	Meaning
0	Ignore the error
1	Retry
2	Abort the program

If you retry, do not change the contents of the DS, BX, CX, or DX registers.

STACK

The stack contains the following:

Top of stack —	IP	System registers from
	CS	Interrupt 24H
	Flags	(fatal error interrupt)
	AX	
	BX	
	CX	
	DX	
	SI	User registers from
	DI	Interrupt 21H
	BP	(disk operation system call)
	DS	
	ES	
	IP	
	CS	
	Flags	

If your error-handling routine does not return to MS-DOS, it should discard the first and last three words from the stack (IP, CS, and Flags at both the top and bottom).

ABSOLUTE DISK READ

ENTRY		RETURN
AL Drive number	INTERRUPT 25H	AL Error code if CF = 1
DS:BX Disk Tranfer Address		
		Flags: CF = 0 if successful
CX Number of sectors		CF = 1 if not successful
DX Beginning relative sector		

Interrupt 25H transfers control directly to the MS-DOS I/O System for a disk read.

The registers must contain the following values:

AL	Drive number (OA, 1B, and so on)
BX	Offset of Disk Transfer Address (from segment address in DS)
CX	Number of sectors to read
DX	Beginning relative sector.

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 that data is read rather than written.

NOTE

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

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags.) Be sure to pop the stack upon return to prevent uncontrolled growth.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains an MS-DOS error code (see Table 2-4 for the codes and their meanings).

Macro Definition:

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

Example:

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

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

ABSOLUTE DISK WRITE

ENTRY →		RETURN →
AL Drive number	INTERRUPT 26H	AL Error code if CF = 1
DS:BX Disk Transfer Address		Flags: CF = 0 if successful CF = 1 if not successful
CX Number of sectors		
DX Beginning relative sector		

Interrupt 26H transfers control directly to the MS-DOS I/O System for a disk write.

The registers must contain the following values:

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

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 that data is written to the disk rather than read from it.

NOTE

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

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags.) Be sure to pop the stack upon return to prevent uncontrolled growth.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains an MS-DOS error code (see Table 2-4 for the codes and their meanings).

Macro Definition:

```
abs_disk_write  macro   disk,buffer,num_sectors, start
                mov     al,disk
                mov     bx,offset buffer
                mov     cx,num_sectors
                mov     dh,start
                int     26K
                endm
```

Example:

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

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

TERMINATE BUT STAY RESIDENT

ENTRY →

CS:DX First byte
following last
byte of code

INTERRUPT 27H

RETURN →

Interrupt 27H keeps a piece of code 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. That is, the program remains resident and is not overlaid by other programs when it terminates.

If an executable file whose extension is .COM ends with this interrupt, it becomes a resident operating system command.

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
```

Function Requests

The standard sequence to call a function request is straightforward.

1. Move any required data into the appropriate registers.
2. Move the function number into AH.
3. Execute Interrupt 21H.

CP/M(R)-COMPATIBLE CALLING SEQUENCE

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

1. Move any required data into the appropriate registers (just as in the standard sequence).
2. Move the function number into the CL register.
3. Execute an intrasegment call to location 5 in the current code segment.

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

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.

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

Table 2-5 lists the function requests.

Table 2-5 MS-DOS Function Requests

FUNCTION NUMBER	FUNCTION NAME
00H	Terminate Program
01H	Read Keyboard and Echo
02H	Display Character
03H	Auxiliary Input
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
07H	Direct Console Input
08H	Read Keyboard
09H	Diplay String
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
0DH	Disk Reset
0EH	Select Disk
0FH	Open File
10H	Close File
11H	Search for First Entry
12H	Search for Next Entry
13H	Delete File
14H	Sequential Read
15H	Sequential Write
16H	Create File
17H	Rename File
19H	Current Disk
1AH	Set Disk Transfer Address
21H	Random Read
22H	Random Write
23H	File Size
24H	Set Relative Record
25H	Set Vector
27H	Random Block Read
28H	Random Block Write
29H	Parse File Name
2AH	Get Date
2BH	Set Date

Table 2-5 MS-DOS Function Requests (cont'd)

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

XENIX-COMPATIBLE CALLS

The hierarchical (that is, tree-structured) directories MS-DOS 2.0 supports, are similar to those found in Microsoft Xenix. (For information on tree-structured directories from the end-user's point of view, refer to the *MS-DOS System User's Guide.*)

The following system calls are Xenix-compatible.

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

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

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

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

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

Table 2-6 lists the directory file attributes and compares them to the attributes for other types of files.

Table 2-6 Directory File Attributes

ATTRIBUTE	MEANING/FUNCTION FOR NON-DIRECTORY FILES	MEANING/FUNCTION FOR DIRECTORIES
Volume ID	Present at the root. Only one file may have this set.	None.
Directory	None.	Indicates that the directory entry is a directory. Cannot be changed with Function 43H.
Read only	Old-FCB create, new create, new open (for write or read/write) will fail.	None.
Archive	Set when file is written. Set/reset via Function 43H.	None.
Hidden/ system	Prevents file from being found in search first/search next operation. New open will fail.	Prevents directory entry from being found. Function 3BH will still work.

TERMINATE PROGRAM

ENTRY → FUNCTION 00H RETURN →

AH 00H

CS Segment address of
Program Segment
Prefix

Function 00H immediately calls Interrupt 20H to terminate a Program. 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.

Exit Address	Offset
Program terminate	0AH
CTRL-C	OEH
Critical error	12H

All file buffers are flushed to disk.

CAUTION

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

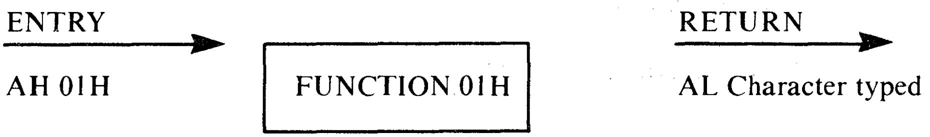
Macro Definition:

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

Example:

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

READ KEYBOARD AND ECHO



ENTRY

AH 01H

FUNCTION 01H

RETURN

AL Character typed

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

Macro Definition:

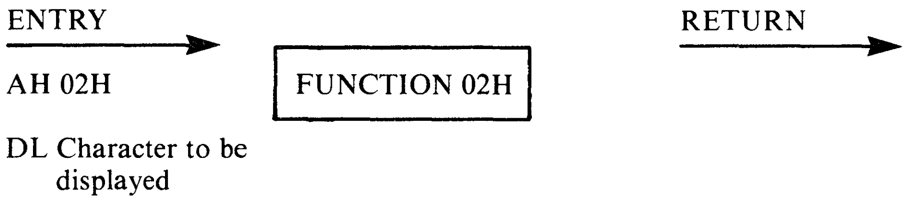
```
read_kbd_and_echo    macro
                     mov      ah, 01H
                     int      21H
                     endm
```

Example:

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

```
func_01H:    read_kbd_and_echo          ;THIS FUNCTION
             print_char    al           ;see Function 05H
             cmp           al,0DH       ;is it a CR?
             jne           func 01H     ;no, print it
             print_char    10           ;see Function 05H
             display_char  10           ;see Function 02H
             jmp           func_01H     ;get another character
```

DISPLAY CHARACTER



ENTRY

AH 02H

DL Character to be displayed

FUNCTION 02H

RETURN

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

Macro Definition:

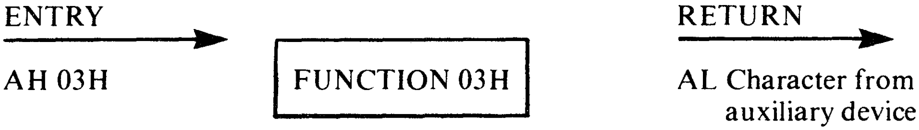
```
display_char  macro   character
              mov     dl,character
              mov     ah,02H
              int     21H
              endm
```

Example:

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

```
func_02H:     read_kbd              ;see Function 08H
              cmp     al,"a"
              jl      uppercase     ;don't convert
              cmp     al,"z"
              jg      uppercase     ;don't convert
              sub     al,20H        ;convert to ASCII code
                                    ;for uppercase
uppercase:    display_char al       ;THIS FUNCTION
              jmp     func_02H:     ;get another character
```

READ KEYBOARD AND ECHO



ENTRY

AH 03H

FUNCTION 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.

Macro Definition:

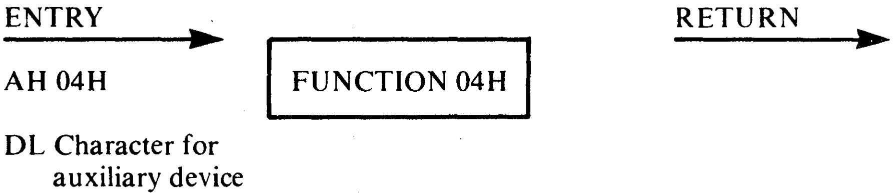
```
aux_input   macro
            mov      ah,03H
            int      21H
            endm
```

Example:

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

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

AUXILIARY OUTPUT



ENTRY

AH 04H

DL Character for
auxiliary device

FUNCTION 04H

RETURN

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

Macro Definition:

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

Example:

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

```
string        db     81 dup(?)                ;see Function 0AH
                     .
                     .
func_04H:     get_string 80,string            ;see Function 0AH
              cmp    string[1],0              ;null string?
              je     continue                 ;yes, all done
              mov    cx, word ptr string[1]   ;get string length
                     mov bx,0                 ;set index to 0
send_it:      aux—output string [bx+2]        ;THIS FUNCTION
              inc    bx                       ;bump index
              loop   send it                  ;send another character
              jmp func_04H                    ;get another string
continue:     .
              .
```

PRINT CHARACTER

ENTRY → FUNCTION 05H RETURN →

AH 05H

DL Character for
printer

Function 05H prints the character in DL. If you press CTRL-C, Interrupt 23H is issued.

Macro Definition:

```
print_char      macro   character
                mov     d1,character
                mov     ah,05H
                int     21H
                endm
```

Example:

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

```
line_num        db      0
                .
                .
func_05H:       mov     cx,60           ;print 60 lines
start_line:     mov     b1,33           ;first printable ASCII
                                        ;character (!)
                add     b1,line_num     ;to offset ne character
                push    cx              ;save number-of-lines counter
                mov     cx,80           ;loop counter for line
print_it:       print_char bl           ;THIS FUNCTION
                inc     b1              ;move to next ASCII character
                cmp     b1,126          ;last printable ASCII
                                        ;character ()
                jl      no_reset        ;not there yet
                mov     b1,33           ;start over with (!)
```

```
no_reset:     loop print_it          ;print another character
              print_char 13          ;carriage return
              print_char 10          ;line feed
              inc     line_num       ;to offset 1st char. of line
              pop     cx             ;restore #-of-lines counter
              loop    start_line;    ;print another line
```

DIRECT CONSOLE I/O

ENTRY		RETURN
AH 06H	FUNCTION 06H	AL If DL = 225 before call
DL 225 = Return character that was typed		Zero not set: No character was ready
Zero clear if no character is typed		Zero set: AL = 0 if character was typed
Return zero set if character is typed		
Not 225 = Display this character		

Function 06H receives input from and sends output to the APC console directly. The processing depends on the value in DL when the function is called.

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

This function does not check for CTRL-C.

Macro Definition:

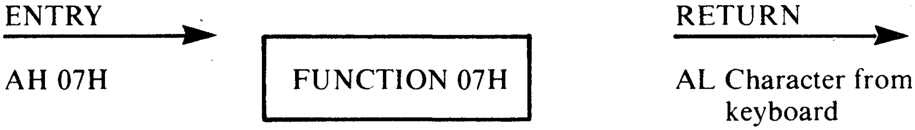
```
dir_console_io    macro switch
                  mov dl,switch
                  mov ah,06H
                  int 21H
                  endm
```

Example:

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

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

DIRECT CONSOLE INPUT



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

Macro Definition:

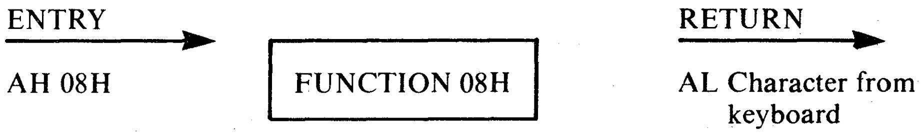
```
dir_console_input   macro
                    mov ah,07H
                    int 21H
                    endm
```

Example:

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

```
password     db      8 dup(?)
prompt       db      "Password: $"      ;see Function 09H for
                                        ;explanation of $
             .
             .
func_07H:    display prompt             ;see Function 09H
             mov     cx,8               ;maximum length of password
             xor     bx,bx              ;so BL can be used as index
get_pass:    dir_console_input          ;THIS FUNCTION
             cmp     al,0DH             ;was it a CR?
             je      continue           ;yes, all done
             mov     password[bx],al    ;no, put character in string
             inc     bx                 ;bump index
             loop    get_pass           ;get another character
continue:    .                          ;BX has length of password+1
             .
```

READ KEYBOARD


ENTRY

AH 08H

FUNCTION 08H

RETURN

AL Character from keyboard

Function 08H waits for a character to be typed, then returns it in AL. If CTRL-C is is pressed, Interrupt 23H is executed. This function does not echo the character on the APC screen. For a keyboard input function that echoes the character, see Function 01H. For one that does not check for CTRL-C, see Function 07H.

Macro Definition:

```
read_kbd   macro
           mov
           ah,O8H
           int 21H
           endm
```

Example:

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

```
password      db      8 dup(?)
prompt        db      "Password: $"        ;see Function 09H
                                           ;for explanation of $
              .

              .
func_08H:      display prompt              ;see Function 09H
              mov     cx,8                 ;maximum length of password
              xor     bx,bx                ;BL can be an index
get_pass:     read_kbd                     ;THIS FUNCTION
              cmp     al,0DH               ;was it a CR?
              je      continue             ;yes, all done
              mov     password[bx],al      ;no, put char. in string
              inc     bx                   ;bump index
              loop    get_pass             ;get another character
continue:     .                            ;BX has length of password+1
              .
```

DISPLAY STRING

ENTRY →

AH 09H

FUNCTION 09H

RETURN →

DS:DX String to be
displayed

Function 09H displays a character string. DX must contain the offset (from the segment address in DS) of a string that ends with "$." The string is displayed (the $ is not displayed).

Macro Definition:

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

Example:

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

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

BUFFERED KEYBOARD INPUT

ENTRY →

AH 0AH

FUNCTION 0AH

RETURN →

DS:DX Input buffer

Function 0AH allows for buffering of keyboard input. DX must contain the offset (from the segment address in DS) of an input buffer. The information in this buffer is the following:

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

Function 0AH waits for characters to be typed. Characters are read from the keyboard and placed in the buffer beginning at the third byte until you press RETURN. If the buffer fills to one less than the maximum, additional characters typed are ignored and ASCII 7 (BEL) is sent to the APC screen until you press RETURN. The string can be edited as it is being entered. If you press CTRL-C, Interrupt 23H is issued.

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

Macro Definition:

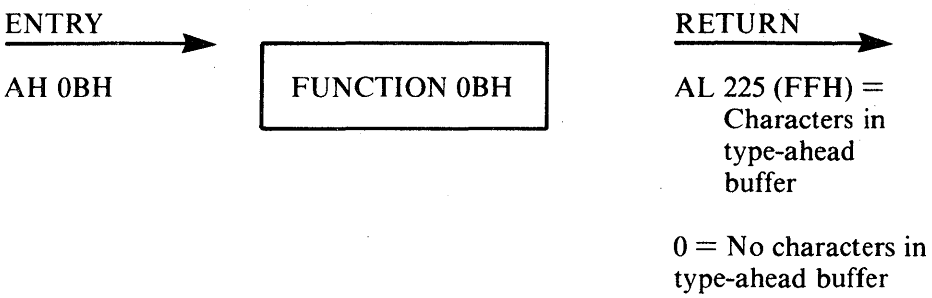
```
get_string  macro   limit,string
            mov     dx,offset string
            mov     string,limit
            mov     ah,0AH
            int     21H
            endm
```

Example:

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

```
buffer              label   byte
max_length          db      ?                       ;maximum length
chars_entered       db      ?                       ;number of chars.
string              db      17 dup (?)              ;16 chars + CR
strings_per_line    dw      0                       ;how many strings
                                                    ;fit on line
crlf                db      13,10,"$"
                    .
                    .
func_0AH:           get_string 17,buffer            ;THIS FUNCTION
                    xor     bx,bx                   ;so byte can be
                                                    ;used as index
                    mov     bl,chars entered        ;get string length
                    mov     buffer [bx+2],"$"       ;see Function 09H
                    mov     a1,50H                  ;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



ENTRY

AH 0BH

FUNCTION 0BH

RETURN

AL 225 (FFH) =
Characters in
type-ahead
buffer

0 = No characters in
type-ahead buffer

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

Macro Definition:

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

Example:

The following program continuously displays the time until you press any key.

```
time         db         "00:00:00.00",13,10,"$"
ten          db         10
             .
             .
func_0BH:    get_time                          ;see Function 2CH
             convert    ch,ten,time            ;see end of chapter
             convert    cl,ten,time[3]         ;see end of chapter
             convert    dh,ten,time[6]         ;see end of chapter
             convert    dl,ten,time[9]         ;see end of chapter
             display    time                   ;see Function 09H
             check_kbd_status                  ;THIS FUNCTION
             cmp        al,FFH                 ;has a key been typed?
             je         all_done               ;yes, go home
             jmp        func_0BH               ;no, keep displaying
                                               ;time
```

FLUSH BUFFER, READ KEYBOARD

ENTRY

AH 0CH

AL 1, 6, 7, 8, or 10 =
The corresponding
function is called
Any other value =
no further
processing

FUNCTION 0CH

RETURN

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

Function 0CH empties the keyboard type-ahead buffer. Further processing depends on the value in AL when the function is called.

- 1, 6, 7, 8, or 10 — The corresponding MS-DOS function is executed.
- Any other value — No further processing; AL returns 0.

Macro Definition:

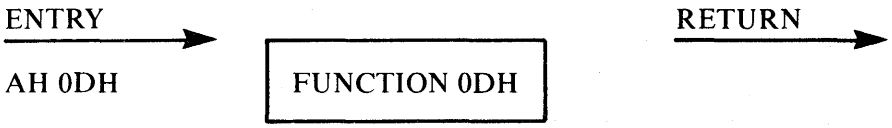
```
flush_and_read_kbd  macro   switch
                    mov     al,switch
                    mov     ah,0CH
                    int     21H
                    endm
```

Example:

The following program both displays and prints characters as they are typed. If you press RETURN, the program sends Carriage Return-Line Feed to both the APC screen and the printer.

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

DISK RESET



ENTRY

AH 0DH

FUNCTION 0DH

RETURN

Function 0DH ensures that the internal buffer cache matches the specified 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. CTRL-C interrupt handlers should call this function.

Macro Definition:

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

Example:

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

SELECT DISK

ENTRY → **FUNCTION 0EH** → RETURN

AH 0EH

DL Drive number
(0 = A, 1 = B,
and so on)

AL Number of
logical drives

Function 0EH allows for selecting a default disk drive. The drive specified in DL (0A, 1B, and so on) is selected as the default disk. The number of drives is returned in AL.

Macro Definition:

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

Example:

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

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

OPEN FILE

ENTRY →		RETURN →
AH 0FH	FUNCTION 0FH	AL 0 = Directory entry found 255 (FFH) = No directory entry found
DS:DX Unopened FCB		

Function 0FH opens a specified file. 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 (1A, 2B, and so on). This lets you change the default disk without interfering with subsequent operations on this file.
- The Current Block field (offset 0CH) is set to zero.
- The Record Size (offset 0EH) is set to the system default of 128.
- The File Size (offset 10H), Date of Last Write (offset 14H), and Time of Last Write (offset 16H) are set from the directory entry.

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

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

Macro Definition:

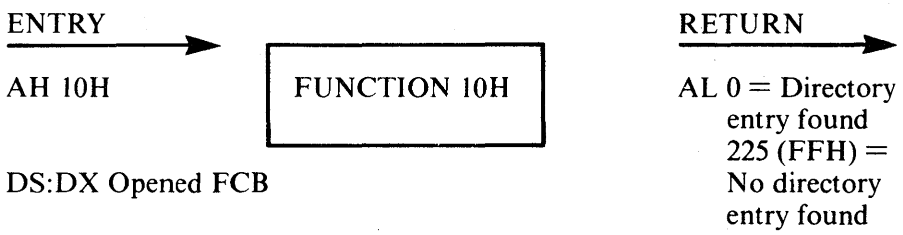
```
open    macro   fcb
        mov     dx,offset fcb
        mov     ah,0FH
        int     21H
        endm
```

Example:

The following program prints the file named TEXTFILE.ASC that is on the diskette 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 CTRL-Z).

```
fcb          db      2,"TEXTFILEASC"
             db      25 dup (?)
buffer       db      128 dup (?)
             .
             .
func_OFH:    set_dta buffer                ;see Function 1AH
             open fcb                      ;THIS FUNCTION
read_line:   read_seq fcb                  ;see Function 14H
             cmp     al,02H                ;end of file?
             je      all_done              ;yes, go home
             cmp     al,00H                ;more to come?
             jg      check_more            ;no, check for partial
                                           ;record
             mov     cx,128                ;yes, print the buffer
             xor     si,si                 ;set index to 0
print_it:    print_char buffer [si]        ;see Function 05H
             inc     si                    ;bump index
             loop    print_it              ;print next character
             jmp     read line             ;read another record
check_more:  cmp     al,03H                ;part. record to print?
             jne     all_done              ;no
             mov     cx,128                ;yes, print it
             xor     si,si                 ;set index to 0
find_eof:    cmp     buffer [si],26        ;end-of-file mark?
             je      all_done              ;yes
             print_char buffer [si]        ;see Function 05H
             inc     si                    ;bump index to next
                                           ;character
             loop    find_eof
all_done:    close fcb                     ;see Function 10H
```

CLOSE FILE



ENTRY

AH 10H

DS:DX Opened FCB

FUNCTION 10H

RETURN

AL 0 = Directory
entry found
225 (FFH) =
No directory
entry found

Function 10H closes a specified file. 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. Thus, Function 10H must be called after a file is changed to update the directory entry.

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

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

Macro Definition:

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

Example:

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

```
message    db     "Not saved in ASCII format",13,10,"$"
fcb        db     2,"MOD1      BAS"
           db     25 dup (?)
buffer     db     128 dup (?)
           .
           .
func_10H:  set_dta buffer      ;see Function 1AH
           open fcb            ;see Function 0FH
           read_seq fcb        ;see Function 14H
           cmp buffer,FFH      ;is first byte FFH?
           jne all_done        ;no
           display message     ;see Function 09H
all_done:  close fcb           ;THIS FUNCTION
```

SEARCH FOR FIRST ENTRY

ENTRY →		RETURN →
AH 11H	FUNCTION 11H	AL 0 = Directory entry found FFH (225) = No directory entry found
DS:DX Unopened FCB		

Function 11H searches for the first entry in a disk directory for a filename. DX must contain the offset (from the segment address in DS) of an unopened FCB. The disk directory is then searched for the first matching name. The name can have the ? wild card character to match any character. To search for hidden or system files, DX must point to the first byte of the Extended FCB prefix.

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

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

Macro Definition:

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

Example:

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

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

             cmp      al,FFH             ;directory entry found?
             je       not_there          ;no
             display  yes                ;see Function 09H
             jmp      continue
not_there:   display  no                 ;see Function 09H
continue:    display  crlf               ;see Function 09H
             .
             .
```

SEARCH FOR NEXT ENTRY

ENTRY →		RETURN →
AH 12H	FUNCTION 12H	AL 0 = Directory entry found FFH (225) = No directory entry found
DS:DX Unopened FCB		

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

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

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

Macro Definition:

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

Example:

The following program displays the number of files on the diskette in drive B.

```
message       db     "No files",10,13,"$"
files         db     0
ten           db     10
fcb           db     2,"???????????"
              db     25 dup (?)
buffer        db     128 dup (?)
              .
              .
func_12H:     set_dta buffer              ;see Function 1AH
              search_first fcb            ;see Function 11H
              cmp     al,FFH              ;directory entry found?
              je      all_done            ;no, no files on disk
              inc     files               ;yes, increment file
                                          ;counter
search_dir:   search_next fcb             ;THIS FUNCTION
              cmp     al,FFH              ;directory entry found?
              je      done                ;no
              inc     files               ;yes, increment file
                                          ;counter
              jmp     search_dir          ;check again
done:         convert files,ten,message   ;see end of chapter
all_done:     display message             ;see Function 09H
```

DELETE FILE

ENTRY → FUNCTION 13H → RETURN

AH 13H

DS:DX Unopened
FCB

AL 0 = Directory
entry found
FFH (225) =
No directory
entry found

Function 13H searches a disk directory for a specified entry to delete it if found. DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for a matching filename. The filename in the FCB can contain the ? wild card character to match any character.

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

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

Macro Definition:

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

Example:

The following program deletes each file on the diskette in drive B that was last written before December 31, 1982.

```
year      dw     1982
month     db     12
day       db     31
files     db     0
ten       db     10
message   db     "NO FILES DELETED.",13,10,"$"
                       ;see Function 09H for
                       ;explanation of $
fcb       db     2,"???????????"
          db     25 dup (?)
```

```
buffer          db      128 dup (?)
                .
                .
func_13H:       set_dta buffer              ;see Function 1AH
                search_first fcb            ;see Function 11H
                cmp     al,FFH              ;directory entry found?
                je      all_done            ;no, no files on disk
compare:        convert_date buffer         ;see end of chapter
                cmp     cx,year             ;next several lines
                jg      next                ;check date in directory
                cmp     dl,month            ;entry against date
                jg      next                ;above & check next file
                cmp     dh,day              ;if date in directory
                jge     next                ;entry isn't earlier.
                delete buffer               ;THIS FUNCTION
                inc     files               ;bump deleted-files
                                            ;counter
next:           search_next fcb             ;see Function 12H
                cmp     al,00H              ;directory entry found?
                je      compare             ;yes, check date
                cmp     files,0             ;any files deleted?
                je      all_done            ;no, display NO FILES
                                            ;message.
                convert files,ten,message   ;see end of chapter
all_done:       display message             ;see Function 09H
```

SEQUENTIAL READ

ENTRY →

AH 14H

DS:DX Opened
FCB

FUNCTION 14H

RETURN →

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

Function 14H reads the next record in a sequence of records. 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 result.

Code	Meaning
0	Read completed successfully.
1	End-of-file, no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

Macro Definition:

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

Example:

The following program displays the file named TEXTFILE.ASC that is on the diskette 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 CTRL-Z).

```
fcb          db    2,“TEXTFILEASC”
             db    25 dup (?)
buffer       db    128 dup (?),“$”
             .
             .
func_14H:    set_dta buffer             ;see Function 1AH
             open fcb                   ;see Function 0FH
read_line:   read_seq fc                ;THIS FUNCTION
```

```
             cmp     al,02H             ;end-of-file?
             je      all_done           ;yes
             cmp     al,02H             ;end-of-file with partial
                                        ;record?
             jg      check_more         ;yes
             display buffer             ;see Function 09H
             jmp     read_line          ;get another record
check_more:  cmp     al,03H             ;partial record in buffer?
             jne     all_done           ;no, go home
             xor     si,si              ;set index to 0
find_eof:    cmp     buffer [si],26     ;is character E0F?
             je      all_done           ;yes, no more to display
             display_char buffer [si]   ;see Function 02H
             inc     si                 ;bump index to next
                                        ;character
             jmp     find_eof           ;check next character
all_done:    close   fcb                ;see Function 10H
```

SEQUENTIAL WRITE

ENTRY →

AH 15H

DS:DX Opened
FCB

FUNCTION 15H

RETURN →

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

Function 15H writes the next record in a sequence of records. 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 OEH 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 result.

Code	Meaning
0	Transfer completed successfully.
1	Disk full; write canceled.
2	Not enough room at the Disk Transfer Address to write one record; write canceled.

Macro Definition:

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

Example:

The following program creates a file named DIR.TMP on the diskette in drive B, which contains the disk number and filename from each directory entry on the diskette. (Disk numbers are assigned as 0 = A, 1 = B, and so on.)

```
record_size   equu          14                     ;offset of Record Size
                                                   ;field in FCB
              .
              .
fcb1          db            2,"DIR TMP"
              db            25 dup (?)
fcb2          db            2,"???????????«
              db            25 dup (?)
buffer        db            128 dup (?)
              .
              .
func_15H:     set_dta       buffer                 ;see Function 1AH
              search_first  fcb2                   ;see Function 11H
              cmp           a1,FFH                 ;directory entry found?
              je            all_done               ;no, no files on disk
              create        fcbl                   ;see Function 16H
              mov           fcbl [record_size],12
                                                   ;set record size to 12
```

```
write_it:     write_seq     fcb1          ;THIS FUNCTION
              search_next   fcb2          ;see Function 12H
              cmp           al,FFH        ;directory entry found?
              je            all_done      ;no, go home
              jmp           write_it      ;yes, write the record
all_done:     close         fcb1          ;see Function 10H
```

CREATE FILE

```
     ENTRY ──────►                               RETURN ──────►

     AH 16H            ┌────────────────┐        AL 00H = Empty
                       │ FUNCTION 16H   │            directory found
                       └────────────────┘            FFH (225) = No
     DS:DX Unopened                                  empty directory
            FCB                                      found
```

Function 16H searches a disk directory for an empty entry or an entry for a specified filename. DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is then searched for the specified entry.

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

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

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

Macro Definition:

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

Example:

The following program creates a file named DIR.TMP on the diskette in drive B, which contains the disk number and filename from each directory entry on the diskette. (Disk numbers are assigned as 0 = A, 1m = B, and so on.)

```
record_size   equ          14                   ;offset of Record Size
                                                ;field of FCB
fcbl          db           2,“DIR TMP”
              db           25 dup (?)
fcb2          db           2,“???????????”
              db           25 dup (?)
buffer        db           128 dup (?)
              .
              .
func_16H:     set_dta      buffer               ;see Function 1AH
              search_first fcb2                 ;see Function 11H
              cmp          al,FFH               ;directory entry
found?        je           all_done             ;no, no files on disk
              create       fcbl                 ;THIS FUNCTION
              mov          fcbl[record_size],12
                                                ;set record size to 12
write_it:     write_seq fcbl                    ;see Function 15H
              search_next fcb2                  ;see Function 12H
              cmp          al,FFH               ;directory entry
found?        je           all_done             ;no, go home
              jmp          write_it             ;yes, write the record
all_done:     close        fcbl                 ;see Function 10H
```

RENAME FILE

ENTRY →		RETURN →
AH 17H	FUNCTION 17H	AL 00H = Directory entry found FFH (225) = No directory entry found or destination already exists
DS:DX Modified FCB		

Function 17H renames the filename in a disk directory entry. DX must contain the offset (from the segment address in DS) of an FCB with the drive number and filename filled in, followed by a second filename at offset 11H. The disk directory is searched for an entry that matches the first filename, which can contain the ? wild card character.

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

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

Macro Definition:

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

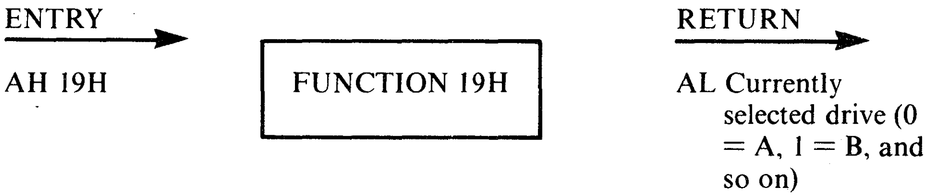
Example:

The following program prompts for the name of a file and a new name, then renames the file.

```
fcb       db    37 dup (?)
prompt1   db    "Filename: $"
prompt2   db    "New name: $"
reply     db    17 dup(?)
crlf      db    13,10,"$"

func_17H:  display     prompt1          ;see Function 09H
           get_string  15,reply         ;see Function 0AH
           display     crlf             ;see Function 09H
           parse       reply[2],fcb     ;see Function 29H
           display     prompt2          ;see Function 09H
           get_string  15,reply         ;see Function 0AH
           display     crlf             ;see Function 09H
           parse       reply[2],fcb=16] ;see Function 29H
           rename      fcb              ;THIS FUNCTION
```

CURRENT DISK



ENTRY

AH 19H

FUNCTION 19H

RETURN

AL Currently selected drive (0 = A, 1 = B, and so on)

Function 19H searches for the currently selected (default) drive. AL returns the drive letter (0 = A, 1 = B, and so on).

Macro Definition:

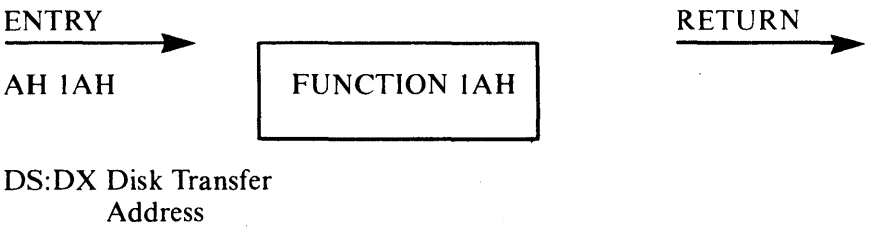
```
current_disk  macro
              mov      ah,19H
              int      21H
              endm
```

Example:

The following program displays the default diskette drive in a two-drive system.

```
message     db          "Current disk is $"   ;see Function 09H
                                              ;for explanation of $
crlf        db          13,10,"$"
            .
            .
func_19H:   display message                   ;see Function 09H
            current_disk                      ;THIS FUNCTION
            cmp         a1,00H                ;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



ENTRY

AH 1AH

DS:DX Disk Transfer
Address

FUNCTION 1AH

RETURN

Function 2AH sets the Disk Transfer Address. DX must contain the offset (from the segment address in DS) of the Disk Transfer Address. Disk transfers cannot wrap around from the end of the segment to the beginning, nor can they overflow into another segment.

NOTE

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

Macro Definition:

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

Example:

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, and so on), then reads and displays the corresponding record from a file named ALPHABET.DAT on the diskette in drive B. The file contains 26 records. Each record is 28 bytes long.

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

```
fcb          db       2, "ALPHABETDAT"
             db       25 dup (?)
buffer       db       34 dup(?),"$"
prompt       db       "Enter letter: $"
crlf         db       13,10,"$"
             .
             .
func_1AH:    set_dta     buffer                    ;THIS FUNCTION
             open        fcb                       ;see Function 0FH
             mov         fcb[record_size],28       ;set record size
get_char:    display     prompt                    ;see Function 09H
             read_kbd_and_echo                     ;see Function 01H
             cmp         al,0DH                    ;just a CR?
             je          all_done                  ;yes, go home
             sub         al,41H                    ;convert ASCII
                                                   ;code to record #
             mov         fcb[relative_record],al   ;set relative record
             display     crlf                      ;see Function 09H
             read_ran    fcb                       ;see Function 21H
             display     buffer                    ;see Function 09H
             display     crlf                      ;see Function 09H
             jmp         get_char                  ;get another character
all_done:    close       fcb                       ;see Function 10H
```

RANDOM READ

ENTRY →		RETURN →
AH 21H	FUNCTION 21H	AL 00H = Read completed successfully 01H = EOF 02H = DTA too small 03H = EOF, partial record
DS:DX Opened FCB		

Function 21H reads the record at a specified address. 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 result.

Code	Meaning
0	Read completed successfully.
1	End-of-file; no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

Macro Definition:

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

Example:

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, and so on), then reads and displays the corresponding record from a file named ALPHABET.DAT on the diskette in drive B. The file contains 26 records. Each record is 28 bytes long.

```
record_size       equu   14   ;offset of Record Size
                              ;field of FCB
relative_record   equ    33   ;offset of Relative Record
                              ;field of FCB
                   .
                   .
fcb             db        2,“ALPHABETDAT
                db        25 dup (?)
buffer          db        34 dup(?),“$”
```

```
prompt      db          "Enter letter: $"
crlf        db          13,10,"$"
                .
                .
func_21H:   set_dta     buffer                  ;see Function 1AH
            open        fcb                     ;see Function 0FH
            mov         fcb[record size],28     ;set record size
get_char:   display     prompt                  ;see Function 09H
            read_kbd_and_echo                   ;see Function 01H
            cmp         al,0DH                  ;just a CR?
            je          all done                ;yes, go home
            sub         al,41H                  ;convert ASCII code
                                                ;to record #
            mov         fcb [relative_record],al ;set relative
                                                ;record
            display     crlf                    ;see Function 09H
            read_ran    fcb                     ;THIS FUNCTION
            display     buffer                  ;see Function 09H
            display     crlf                    ;see Function 09H
            jmp         get_char                ;get another char.
all_done:   close fcb                           ;see Function 10H
```

RANDOM WRITE

ENTRY →		RETURN →
AH 22H	FUNCTION 22H	AL 00H = Write completed successfully 01H = Disk full 02H = DTA too small
DS:DX Opened FCB		

Function 22H writes a specified 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 written from the Disk Transfer Address. If the record size is smaller than a sector (512 bytes), the records are buffered until a sector is ready to write.

AL returns a code that describes the processing result.

Code	Meaning
0	Write completed successfully.
1	Disk is full.
2	Not enough room at the Disk Transfer Address to write one record; write canceled.

Macro Definition:

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

Example:

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, and so on), then reads and displays the corresponding record from a file named ALPHABET.DAT on the diskette in drive B. After displaying the record, it prompts the user to enter a changed record. If you type a new record, it is written to the file; if you just press RETURN, the record is not replaced. The file contains 26 records. Each record is 28 bytes long.

```
record_size       equu    14   ;offset of Record Size
                               ;field of FCB
relative_record   equ     33   ;offset of Relative Record
                               ;field of FCB
            .
            .
fcb         db   2,“ALPHABETDAT”
            db   25 dup (?)
buffer      db   26 dup(?),13,10,“$”
prompt1     db   “Enter letter: $”
prompt2     db   “New record (RETURN for no change): $”
crlf        db   13,10,“$”
reply       db   28 dup (32)
blanks      db   26 dup (32)
            .
            .
```

```
func_22H:    set_dta     buffer                   ;see Function 1AH
             open        fcb                      ;see Function 0FH
             mov         fcb[record size],32      ;set record size
get_char:    display     prompt1                  ;see Function 09H
             read_kbd_and_echo                    ;see Function 01H
             cmp         al,0DH                   ;just a CR?
             je          all_done                 ;yes, go home
             sub         al,41H                   ;convert ASCII
                                                  ;code to record #
             mov         fcb[relative_record],al
                                                  ;set relative record
             display     crlf                     ;see Function 09H
             read_ran    fcb                      ;THIS FUNCTION
             display     buffer                   ;see Function 09H
             display     crlf                     ;see Function 09H
             display     prompt2                  ;see Function 09H
             get_string  27,reply                 ;see Function 0AH
             display     crlf                     ;see Function 09H
             cmp         reply[1],0               ;was anything typed
                                                  ;besides CR?
             je          get_char                 ;no
                                                  ;get another char.
             xor         bx,bx                    ;to load a byte
             mov         bl,reply[1]              ;use reply length as
                                                  ;counter
             move_string blanks,buffer,26         ;see chapter end
             move_string reply[2],buffer,bx       ;see chapter end
             write_ran fcb                        ;THIS FUNCTION
             jmp         get_char                 ;get another character
all_done:    close       fcb                      ;see Function 10H
```

FILE SIZE

ENTRY →		RETURN →
AH 23H	FUNCTION 23H	AL 00H = Directory entry FFH (225) = No directory entry found
DS:DX Opened FCB		

Function 23H searches for the size of a specified file. DX must contain the offset (from the segment address in DS) of an unopened FCB. You must set the Record Size field (offset OEH) to the proper value before calling this function. The disk directory is searched for the first matching entry.

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

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

NOTE

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

Macro Definition:

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

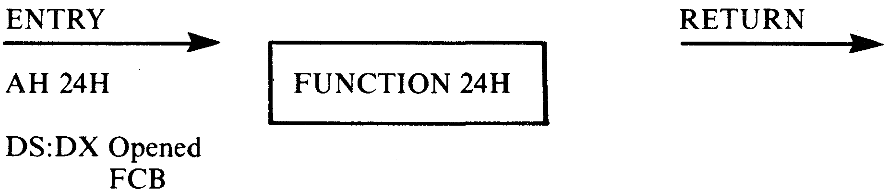
Example:

The following program prompts for the name of a file, opens the file to fill in the Record Size field of the FCB, issues a File Size system call, and displays the file size and number of records in hexadecimal.

```
fcb          db   37 dup (?)
prompt       db   “File name: $”
msg1         db   “Record length: ”,13,10,“$”
msg2         db   “Records: ”,13,10,“$”
crlf         db   13,10,“$”
reply        db   17 dup(?)
sixteen      db   16
                  .
                  .
```

```
func_23H:     display     prompt                      ;see Function 09H
              get_string 17,reply                     ;see Function 0AH
              cmp         reply[1],0                  ;just a CR?
              jne         get_length                  ;no, keep going
              jmp         all done                    ;yes, go home
get_length:   display     crlf                        ;see Function 09H
              parse       reply[2],fcb                ;see Function 29H
              open        fcb                         ;see Function 0FH
              file_size   fcb                         ;THIS FUNCTION
              mov         si,33                       ;offset to Relative
                                                      ;Record field
              mov         di,9                        ;reply in msg_2
convert_it:   cmp         fcb[si],0                   ;digit to convert?
              je          show_it                     ;no, prepare message
              convert fcb[si],sixteen,msg_2 [di]
              inc         si                          ;bump n-o-r index
              inc         di                          ;bump message index
              jmp         convert it                  ;check for a digit
show_it:      convert     fcb [14],sixteen,msg_1[15]
              display     msg_1                       ;see Function 09H
              display     msg_2                       ;see Function 09H
              jmp         func_23H                    ;get a filename
all_done:     close       fcb                         ;see Function 10H
```

SET RELATIVE RECORD


ENTRY

AH 24H

DS:DX Opened FCB

FUNCTION 24H

RETURN

Function 24H sets the relative record address for a random read and write operation. DX must contain the offset (from the segment address in DS) of an opened FCB. The Relative Record field (offset 21H) is set to the same file address as the Current Block (offset 0CH) and Current Record (offset 20H) fields.

Macro Definition:

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

Example:

The following program copies a file using the Random Block Read and Random Block Write system calls (Functions 27H and 28H). It speeds the copy by setting the record length equal to the file size and the record count to 1, and using a buffer of 32K bytes. It positions the file pointer by setting the Current Record field (offset 20H) to 1 and using Set Relative Record to make the Relative Record field (offset 21H) point to the same record as the combination of the Current Block (offset 0CH) and Current Record (offset 20H) fields.

```
current_record    equu    32   ;offset of Current Record
                               ;field of FCB
file_size         equ     16   ;offset of File Size
                               ;field of FCB
              .
              .
fcb         db    37 dup (?)
filename    db    17 dup(?)
prompt1     db    "File to copy: $"      ;see Function 09H for
prompt2     db    "Name of copy: $"     ;explanation of $
crlf        db    13,10,"$"

file_length  dw          ?
buffer       db          32767 dup(?)
func_24H:    set_dta     buffer                     ;see Function 1AH
             display     prompt1                    ;see Function 09H
             get_string  15,filename                ;see Function 0AH
             display     crlf                       ;see Function 09H
             parse       filename[2],fcb            ;see Function 29H
             open        fcb                        ;see Function 0FH
             mov         fcb[current_record],0      ;set Current Record
                                                    ;field
             set_relative_record fcb                ;THIS FUNCTION
             mov         ax,word ptr fcb[file size] ;get file size
```

```
        mov           file_length,ax               ;save it for
                                                   ;ran_block_write
        ran_block_read fcb,1,ax                    ;see Function 27H
        display       prompt2                      ;see Function 09H
        get_string 15,filename                     ;see Function 0AH
        display       crlf                         ;see Function 09H
        parse         filename[2],fcb              ;see Function 29H
        create        fcb                          ;see Function 16H
        mov           fcb[current record],0        ;set Current Record
                                                   ;field
        set_relative_record fcb                    ;THIS FUNCTION
        mov           ax,file_length               ;get original file
                                                   ;length
        ran_block_write fcb,1,ax                   ;see Function 28H
        close         fcb                          ;see Function 10H
```

SET VECTOR

ENTRY → FUNCTION 25H RETURN →

AH 25H

AL Interrupt number

DS:DX Interrupt-
handling
routine

Function 25H should be used to set a particular interrupt vector. The operating system can then manage the interrupts on a per-process basis.

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

Macro Definition:

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

Example:

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

RANDOM BLOCK READ

ENTRY →		RETURN →
AH 27H	FUNCTION 27H	AL 00H = Read completed successfully 01H = EOF 02H = End of segment 03H = EOF
DS:DX Opened FCB		
CX Number of blocks to read		CX Number of blocks read

Function 27H reads a specified block of records. 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 result.

Code	Meaning
0	Read completed successfully.
1	End-of-file; no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

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

Macro Definition:

```
ran_block_read   macro    fcb,count,rec_size
                 mov      dx,offset fcb
                 mov      cx,count
                 mov      word ptr fcb[14],rec_size
                 mov      ah,27H
                 int      21H
                 endm
```

Example:

The following program copies a file using the Random Block Read system call. It speeds the copy by specifying a record count of 1 and a record length equal to the file size, and using a buffer of 32K bytes. The file is read as a single record. (Compare to the sample program for Function 28H, which specifies a record length of 1 and a record count equal to the file size.)

```
current_record   equ   32   ;offset of Current Record field
file_size        equ   16   ;offset of File Size field
               .
               .
fcb         db   37 dup (?)
filename    db   17 dup(?)
prompt1     db   "File to copy: $"     ;see Function 09H for
prompt2     db   "Name of copy: $"     ;explanation of $
```

```
crlf          db    13,10,"$"
file_length   dw    ?
buffer        db    32767 dup(?)
                .
                .
func_27H:     set_dta       buffer                ;see Function 1AH
              display       prompt1               ;see Function 09H
              get_string 15,filename              ;see Function 0AH
              display       crlf                  ;see Function 09H
              parse         filename[2],fcb       ;see Function 29H
              open          fcb                   ;see Function 0FH
              mov           fcb[current_record],0    ;set Current
                                                  ;Record field
              set_relative_record fcb
                                                  ;see Function 24H
              mov ax, word ptr fcb[file_size]
                                                  ;get file size
              mov           file_length,ax        ;save it for
                                                  ;ran_block_write
              ran_block_read fcb,1,ax             ;THIS FUNCTION
              display       prompt2               ;see Function 09H
              get_string 15,filename              ;see Function 0AH
              display       crlf                  ;see Function 09H
              parse         filename[2],fcb       ;see Function 29H
              create        fcb                   ;see Function 16H
              mov           fcb[current record],0 ;set Current Record
                                                  ;field
              set_relative_record fcb             ;see Function 24H
              mov           ax, file_length       ;get original file
                                                  ;size
              ran_block_write fcb,1,ax            ;see Function 28H
              close         fcb                   ;see Function 10H
```

RANDOM BLOCK WRITE

ENTRY →		RETURN →
AH 28H	FUNCTION 28H	AL 00H = Write completed successfully 01H = Disk full 02H = End of segment
DS:DX Opened FCB		
Number of blocks to write (0 = set File Size field)		CX Number of blocks written

Function 28H writes a specified block of records. 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 result.

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 21 H) fields are set to address the next record.

Macro Definition:

```
ran_block_write  macro   fcb,count,rec_size
                 mov     dx,offset fcb
                 mov     cx,count
                 mov     word ptr fcb[14],rec_size
                 mov     ah,28H
                 int     21H
                 endm
```

Example:

The following program copies a file using the Random Block Read and Random Block Write system calls. It speeds the copy by specifying a record count equal to the file size and a record length of 1, and using a buffer of 32K bytes. The file is copied quickly with one disk access each to read and write. (Compare to the sample program for Function 27H, which specifies a record count of 1 and a record length equal to file size.)

```
current_record   equu    32   ;offset of Current Record field
file_size        equ     16   ;offset of File Size field
               .
               .
fcb            db    37 dup (?)
filename       db    17 dup(?)
prompt1        db    "File to copy: $"     ;see Function 09H for
prompt2        db    "Name of copy: $"     ;explanation of $
crlf           db    13,10,"$"
num_recs       dw    ?
buffer         db    32767 dup(?)
               .
               .
func_28H:      set_dta      buffer                   ;see Function 1AH
               display      prompt1                  ;see Function 09H
               get_string   15,filename              ;see Function 0AH
               display      crlf                     ;see Function 09H
               parse        filename[2],fcb          ;see Function 29H
               open         fcb                      ;see Function 0FH
               mov          fcb [current record],0
                                                     ;set Current Record
                                                     ;field
```

```
set_relative_record fcb              ;see Function 24H
mov         ax, word ptr fcb[file_size]
                                     ;get file size
mov         num_recs,ax              ;save it for
                                     ;ran_block_write
ran_block_read fcb,num_recs,1        ;THIS FUNCTION
display     prompt2                  ;see Function 09H
get_string  15,filename              ;see Function 0AH
display     crlf                     ;see Function 09H
parse       filename [2],fcb         ;see Function 29H
create      fcb                      ;see Function 16H
mov         fcb[current record],0    ;set Current
                                     ;Record field
set_relative_record fcb              ;see Function 24H
mov         ax, file_length          ;get size of original
ran_block_write fcb,num_recs,1       ;see Function 28H
close       fcb                      ;see Function 10H
```

PARSE FILE NAME

ENTRY →

AH 29H

AL Controls parsing

DS:DI String to parse

FUNCTION 29H

RETURN →

AL 00H = No
wild characters

01H =Wild card
characters used
FFH (225) =
Drive letter

DS:SI First byte
past string
that was
parsed

ES:DI Unopened
FCB

Function 29H parses a command line (string) for the filename. SI must contain the offset (to the segment address in DS) of a string (command line) to parse. DI must contain the offset (to the segment address in ES) of an unopened FCB. The string is parsed for a filename of the form d:filename.ext. If one is found, a corresponding unopened FCB is created at ES:DI.

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

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

If the filename or extension includes an asterisk (*), all remaining characters in the name or extension are set to question mark (?).

The following are legal filename separators:

: . ; , + / ” [ ] space tab

Filenames in a string are ended by filename terminators. Filename terminators can be any of the filename separators or any control character. A filename cannot contain a filename terminator. If one is encountered, parsing stops.

If the string contains a valid filename,

- 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.

- DS:SI point to the first character following the string that was parsed.
- ES:DI point to the first byte of the unopened FCB.

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

Macro Definition:

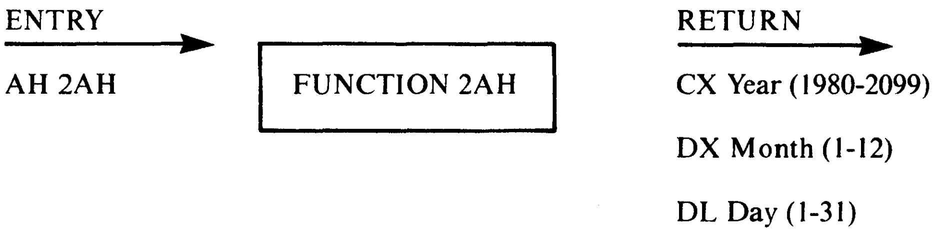
```
parse   macro   string,fcb
        mov     si,offset string
        mov     di,offset fcb
        push    es
        push    ds
        pop     es
        mov     al,0FH                  ;bits 0, 1, 2, 3 on
        mov     ah,29H
        int     21H
        pop     es
        endm
```

Example:

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

```
fcb       db   37 dup (?)
prompt    db   “Filename: $”
reply     db   17 dup(?)
yes       db   “FILE EXISTS”,13,10,“$”
no        db   “FILE DOES NOT EXIST”,13,10,“$”
               .
               .
func_29H:      display        prompt        ;see
               get_string     15,reply      ;see Function 0AH
               parse          reply[2],fcb  ;THIS FUNCTION
               search_first   fcb           ;see Function 11H
               cmp            al,FFH        ;dir. entry found?
               je             not_there     ;no
               display        yes           ;see Function 09H
               jmp            continue
not_there:     display        no
continue:      .
               .
```

GET DATE



ENTRY

AH 2AH

FUNCTION 2AH

RETURN

CX Year (1980-2099)

DX Month (1-12)

DL Day (1-31)

Function 2AH returns the current date set in the operating system as binary numbers in CX and DX.

Macro Definition:

```
get_date    macro
            mov     ah,2AH
            int     21H
            endm
```

Example:

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date.

```
month       db      31,28,31,30,31,30,31,31,30,31,30,31
            .
            .
func_2AH:   get_date                ;see above
            inc     dl              ;increment day
            xor     bx,bx           ;so BL can be used as index
            mov     bl,dh           ;move month to index register
            dec     bx              ;month table starts with 0
            cmp     dl,month[bx]    ;past end of month?
            jle     month_ok        ;no, set the new date
            mov     dl,1            ;yes, set day to 1
            inc     dh              ;and increment month
            cmp     dh,12           ;past end of year?
page        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

ENTRY →

AH 2BH

CS Year (1980-2099)

DH Month (1-12)

DL Day (1-31)

FUNCTION 2BH

RETURN →

Al 00H = Date
was valid
FFH (225) =
Date was invalid

Function 2BH sets the system date. Registers CX and DX must contain a valid date in binary.

If the date is valid, the date is set and AL returns 0. If the date is not valid, the function is canceled and AL returns FFH (255).

Macro Definition:

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

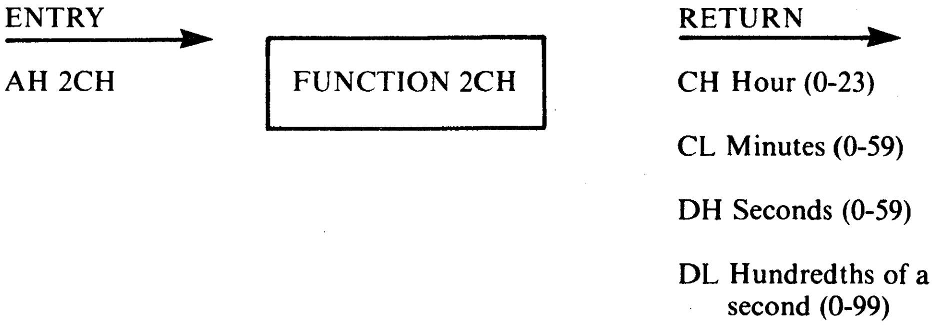
Example:

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date.

```
month       db       31,28,31,30,31,30,31,31,30,31,30,31
            .
            .
func_2BH:   get_date                   ;see Function 2AH
            inc      dl                ;increment day
            xor      bx,bx             ;so BL can be used as index
            mov      bl,dh             ;move month to index register
            dec      bx                ;month table starts with 0
            cmp      dl,month[bx]      ;past end of month?
            jle      month_ok          ;no, set the new date
```

```
                mov     dl,1            ;yes, set day to 1
                inc     dh              ;and increment month
                cmp     dh,12           ;past end of year?
                jle     month_ok        ;no, set the new date
                mov     dh,1            ;yes, set the month to 1
                inc     cx              ;increment year
month_ok:       set_date cx,dh,dl       ;THIS FUNCTION
```

GET TIME



ENTRY

AH 2CH

FUNCTION 2CH

RETURN

CH Hour (0-23)

CL Minutes (0-59)

DH Seconds (0-59)

DL Hundredths of a second (0-99)

Function 2CH returns the current time set in the operating system as binary numbers in CX and DX.

Macro Definition:

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

Example:

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

```
time        db   "00:00:00.00",13,10,"$"
ten         db   10
            .
            .
func_2CH:   get_time                    ;THIS FUNCTION
            convert   ch,ten,time       ;see end of chapter
            convert   cl,ten,time[3]    ;see end of chapter
```

```
        convert     dh,ten,time[6]    ;see end of chapter
        convert     dl,ten,time[9]    ;see end of chapter
        display     time              ;see Function 09H
        check_kbd_status              ;see Function 0BH
        cmp         al,FFH            ;has a key been pressed?
        je          all_done          ;yes, terminate
        jmp         func_2CH          ;no, display time
```

SET TIME

ENTRY →

AH 2DH

CH Hours (0-23)

CL Minutes (0-59)

DH Seconds (0-59)

DL Hundredths of a
second (0-99)

FUNCTION 2DH

RETURN →

AL 00H = Time
was valid
FFH (225) =
Time was
invalid

Function 2DH sets the system time. Registers CX and DX must contain a valid time in binary.

If the time is valid, the time is set and AL returns 0. If the time is not valid, the function is canceled and AL returns FFH (255).

Macro Definition:

```
set_time    macro    hour,minutes,seconds,hundredths
            mov      ch,hour
            mov      cl,minutes
            mov      dh,seconds
            mov      dl,hundredths
            mov      ah,2DH
            int      21H
            endm
```

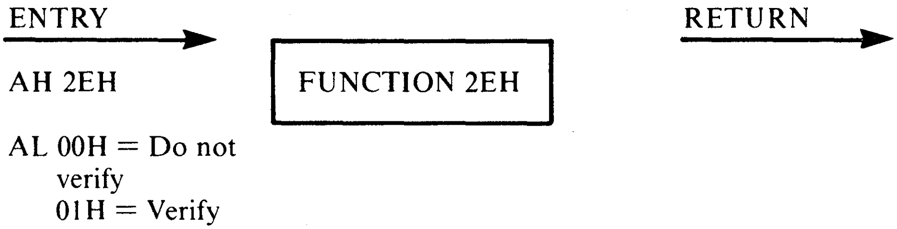
Example:

The following program sets the system clock to 0 and continuously displays the time. When a character is typed, the display freezes. When another character is typed, the clock is reset to 0 and the display starts again.

```
time          db        "00:00:00.00",13,10,"$"
ten           db        10
                 .
                 .
func_2DH:     set_time        0,0,0,0     ;THIS FUNCTION
read_clock:   get_time                    ;see Function 2CH

              convert         ch,ten,time      ;see end of chapter
              convert         cl,ten,time[3]   ;see end of chapter
              convert         dh,ten,time[6]   ;see end of chapter
              convert         dl,ten,time[9]   ;see end of chapter
              display         time             ;see Function 09H
              dir_console_io  FFH              ;see Function 06H
              cmp             al,00H           ;was a char. typed?
              jne             stop             ;yes, stop the timer
              jmp             read_clock       ;no keep timer on
stop:         read_kbd                         ;see Function 08H
              jmp             func_2DH         ;keep displaying time
```

SET/RESET VERIFY FLAG



ENTRY

AH 2EH

AL 00H = Do not
verify
01H = Verify

FUNCTION 2EH

RETURN

Function 2EH sets and resets the verify flag for a write. AL must be either 1 (verify after each disk write) or 0 (write without verifying). MS-DOS checks this flag each time it writes to a disk.

The flag is normally off. You may wish to turn it on when writing critical data to disk. Because disk errors are rare and verification slows writing, you will probably want to leave it off at other times.

Macro Definition:

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

Example:

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

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

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

```
        pop     cx                      ;restore counter
        loop    copy                    ;do it again
        verify  off
```

GET DISK TRANSFER ADDRESS

ENTRY → FUNCTION 2FH → RETURN

AH 2FH

FUNCTION 2FH

ES:BX Points to Disk Transfer Address

Function 2FH returns the DMA transfer address.

Error returns:

None.

Example:

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

GET DOS VERSION NUMBER

ENTRY → FUNCTION 30H → RETURN

AH 30H

FUNCTION 30H

AL Major version number

AH Minor version number

Function 30H returns the MS-DOS version number. On return, AL:AH will be the two-part version designation; that is, for MS-DOS 1.28, AL would be 1 and AH would be 28. For pre-1.28 DOS, AL = 0. Note that version 1.1 is the same as 1.10, not the same as 1.01.

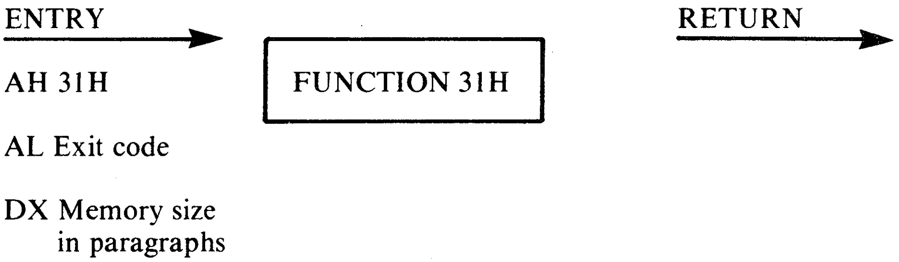
Error returns:

None.

Example:

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

KEEP PROCESS



ENTRY

AH 31H

AL Exit code

DX Memory size
in paragraphs

FUNCTION 31H

RETURN

Function 31H terminates the current process and attempts to set the initial allocation block to a specific size in paragraphs. It will not free up any other allocation blocks belonging to that process.

The exit code passed in AX is retrievable by the parent via Function 4DH.

Error returns:

None.

Example:

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

CTRL-CHECK

ENTRY →

AH 33H

AL Function 00H =
Request current
state
01H = Set state

DL (if setting)
00H = Off
01H = On

FUNCTION 33H

RETURN →

DL 00H = Off,
01H = On

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

Error return:

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

Example:

```
mov     dl,val
mov     ah,33H
mov     al,func
int     21H
        ; If al was 0, then dl has the current value
        ;of the CTRL-C check
```

GET INTERRUPT VECTOR

ENTRY →

AH 35H

AL Interrupt number

FUNCTION 35H

RETURN →

ES:BX Pointer to
interrupt
routine

Function 35H returns the interrupt vector associated with an interrupt.

Error returns:

None.

Example:

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

GET DISK FREE SPACE

ENTRY		RETURN
AH 36H	FUNCTION 36H	BX Available clusters
DL Drive (0 = default, 1 = A, and so on)		DX Clusters per drive
		AX FFFF if drive number is invalid; otherwise, sectors per cluster

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

Error return:

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

Example:

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

RETURN COUNTRY-DEPENDENT INFORMATION

ENTRY →		RETURN →
AH 38H	FUNCTION 38H	Carry set: AX = 2 file not found Carry not set: DS:SX filled with country data
DS:DX Pointer to 32-byte memory area		
AL Function code; in MS-DOS 2.0, must be 0		

Function 38H returns country-dependent information. The value passed in AL is either 0 (for current country) or other country code. Country codes are typically the international telephone prefix code for the country.

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

NOTE

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

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

WORD Date/time format
5 BYTE ASCIZ string Currency symbol
2 BYTE ASCIZ string Thousands separator
2 BYTE ASCIZ string Decimal separator

2 BYTE ASCIZ string Date separator
2 BYTE ASCIZ string Time separator
1 BYTE Bit field
1 BYTE Currency Places
1 BYTE Time format
DWORD Case Mapping call
2 BYTE ASCIZ string Data List separator

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

The date/time format has the following values:

Value	Format	
0	USA standard	h:m:s m/d/y
1	Europe standard	h:m:s d/m/y
2	Japan standard	y/m/d h:m:s

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

Bit 0 = 0 If currency symbol precedes the currency amount.

= 1 If currency symbol comes after the currency amount.

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

The time format has the following values:

Value	Format
0	12 hour time
1	24 hour time.

The Currency Places field indicates the number of places that appear after the decimal point on currency amounts.

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

Error return:

AX
2 = File not found. The country passed in AL was not found (no table for specified country).

Example:

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

CREATE SUB-DIRECTORY

ENTRY

AH 39H

DS:DX Pointer to path-name

FUNCTION 39H

RETURN

Carry set:
AX 3 = Path not found
5 = Access denied

Carry not set: No error

Given a pointer to an ASCIZ name, Function 39H creates a new directory entry at the end.

Error returns:

AX

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

Example:

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

REMOVE A DIRECTORY ENTRY

ENTRY		RETURN
AH 3AH	FUNCTION 3AH	Carry set: AX 3 = Path not found 5 = Access denied 16 = Current directory Carry not set: No error
DX Pointer to pathname		

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

Error returns:

AX

3 = Path not found. The path specified was invalid or not found.
5 = Access denied. The path specified was not empty, not a directory, the root directory, or contained invalid information.
16 = Current directory. The path specified was the current directory on a drive.

Example:

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

CHANGE THE CURRENT DIRECTORY

ENTRY		RETURN
AH 3BH	FUNCTION 3BH	Carry set: AX = Path not found
DS:DX Pointer to pathname		Carry not set: No error

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

Error return:

AX

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

Example:

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

CREATE A FILE

ENTRY		RETURN
AH 3CH	FUNCTION 3CH	Carry set: AX 5 = Access denied
DS:DX Pointer to pathname		3 = Path not found 4 = Too many open files
CX File attribute		Carry not set: AX is handle number

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

Error returns:

AX

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

Example:

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

OPEN A FILE

ENTRY →

AH 3DH

AL Access:
0 = File opened for reading
1 = File opened for writing
2 = File opened for both reading and writing

FUNCTION 3DH

RETURN →

Carry set:
AX 12 = Invalid access
2 = File not found
5 = Access denied
4 = Too many open files

Carry not set: AX is handle number

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

The following values are allowed:

Access	Function
0	File is opened for reading
1	File is opened for writing
2	File is opened for both reading and writing.

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

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

Error returns:

AX

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

Example:

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

CLOSE A FILE HANDLE

ENTRY →		RETURN →
AH 3EH	FUNCTION 3EH	Carry set: AX 6 = Invalid handle
BX File handle		Carry not set: No error

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

Error return:

AX

6 = Invalid handle. The handle passed in BX was not currently open.

Example:

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

READ FROM FILE/DEVICE

ENTRY →		RETURN →
AH 3FH	FUNCTION 3FH	Carry set: AX Number of bytes read 6 = Invalid handle 5 = Error set
DS:DX Pointer to buffer		
CX Bytes to read		Carry not set: AX = number of bytes read
BX File handle		

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

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

Error returns:

AX

6 = Invalid handle. The handle passed in BX was not currently open.
5 = Access denied. The handle passed in BX was opened in a mode that did not allow reading.

Example:

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

WRITE TO A FILE/DEVICE

ENTRY		RETURN
AH 40H	FUNCTION 40H	Carry set: AX Number of bytes written 6 = Invalid handle 5 = Access
DS:DX Pointer to buffer		
BX File handle		Carry not set: A = Number of bytes written

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

The write system call with a count of zero (CX = 0) will truncate the file at the current position.

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

Error returns:

AX

6 = Invalid handle. The handle passed in BX was not currently open.
5 = Access denied. The handle was not opened in a mode that allowed writing.

Example:

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

DELETE A DIRECTORY ENTRY

ENTRY →

AH 41H

DS:DX Pointer to pathname

FUNCTION 41H

RETURN →

Carry set:
AX 2 = File not found
5 = Access denied

Carry not set: No error

Function 41H removes the directory entry associated with a filename. If the file is currently open on another handle, then no removal will take place.

Error returns:

AX

2 = File not found. The path specified was invalid or not found.
5 = Access denied. The path specified was a directory or read-only.

Example:

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

MOVE FILE POINTER

ENTRY →

AH 42H

CX:DX Distance to
move in
bytes

AL Method of moving

BX File handle

FUNCTION 42H

RETURN →

Carry set:
AX 6 = Invalid
handle
1 = Invalid
function

Carry not set:
DX:AX = New
pointer location

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

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

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

Error returns:

AX

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

Example:

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

CHANGE ATTRIBUTES

ENTRY		RETURN
AH 43H	FUNCTION 43H	Carry set: AX 3 = Path not found 5 = Access denied 1 = Invalid function
DS:DX Pointer to pathname		
CS (if AL = 01)		
AL Function 01 = set to CX; 00 = return in CX		Carry not set: CX attributes (if AL = 00)

Given an ASCIZ name, Function 43H will set/get the attributes of the file to those given in CX.

A function code is passed in AL.

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

Error returns:

AX

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

Example:

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

I/O CONTROL FOR DEVICES

ENTRY

AH 44H

BX Handle

BL Drive for calls
AL = 4,5 (0 = default,
1 = A, and so on)

DS:DX Data for
buffer

CX Bytes to read or
write

AL Function code

FUNCTION 44H

RETURN

Carry set:
AX 6 = Invalid
handle
1 = Invalid function
13 = Invalid date
5 = Access denied

Carry not set:
AL if 2, 3, 4, 5 then
AX = Count
transferred
if 6, 7 then 00 =
Not ready;
FF = ready

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

The following values are allowed for the function:

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

Function 44H can be used to get information about device channels. Calls can be made on regular files, but only calls 0, 6 and 7 are defined in that case (AL = 0, 6, 7). All other calls return an invalid function error.

Calls AL = 0 and AL = 1

The bits of DX are defined as follows for calls AL0 and AL1. Note that the upper byte MUST be zero on a set call.

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

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

If ISDEV = 1
EOF = 0 if End Of File on input
RAW = 1 if this device is in Raw mode
= 0 if this device is cooked
ISCLK = 1 if this device is the clock device
ISNUL = 1 if this device is the null device
ISCOT = 1 if this device is the console output
ISCIN = 1 if this device is the console input
SPECL = 1 if this device is special
CTRL = 0 if this device cannot do control strings via calls
AL = 2 and AL = 3
CTRL = 1 if this device can process control strings via calls
AL = 2 and AL = 3
NOTE that this bit cannot be set.
If ISDEV = 0
EOF = 0 if channel has been written
Bits 0-5 are the block device number for the channel
(0 = A, 1 = B, and so on)

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

Calls AL = 0 through AL = 5

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

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

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

Calls AL = 6 and AL = 7

These two calls allow you to check if a file handle is ready for input or output. These calls are intended for checking the status of handles open to a device, but they can also be used to check the status of a handle open to a disk file. The statuses are defined as follows:

Input:

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

Output:

Always ready (even if disk is full).

CAUTION

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

Error returns:

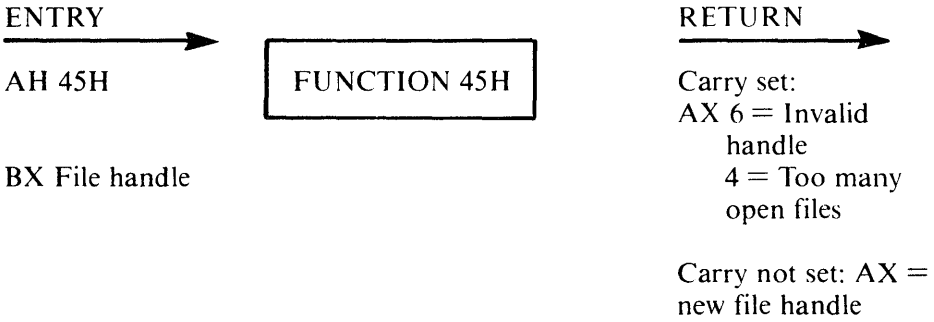
AX

6 = Invalid handle. The handle passed in BX was not currently open.
1 = Invalid function. The function passed in AL was not in the range 0:7.
13 = Invalid data.
5 = Access denied (calls AL4 through AL7).

Example:

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

DUPLICATE A FILE HANDLE



ENTRY

AH 45H

BX File handle

FUNCTION 45H

RETURN

Carry set:
AX 6 = Invalid
handle
4 = Too many
open files

Carry not set: AX =
new file handle

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

Error returns:

AX

6 = Invalid handle. The handle passed in BX was not currently open.
4 = Too many open files. There were no free handles available in the current process or the internal system tables were full.

Example:

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

FORCE A DUPLICATE OF A FILE HANDLE

ENTRY →		RETURN →
AH 46H	FUNCTION 46H	Carry set: AX 6 = Invalid handle 4 = Too many open files
BX Existing file handle		
CX New file handle		Carry not set: No error

Function 46H takes an already opened file handle and returns a new one that refers to the same file at the same position.

Error returns:

AX

6 = Invalid handle. The handle passed in BX was not currently open.
4 = Too many open files. There were no free handles available in the current process or the internal system tables were full.

Example:

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

RETURN TEXT OF CURRENT DIRECTORY

ENTRY →		RETURN →
AH 47H	FUNCTION 47H	Carry set: AX 15 = Invalid drive
DS:SI Pointer to 64-byte memory area		Carry not set: No error
DL Drive number		

Function 47H returns the current directory for a particular drive. The directory is root-relative and does not contain the drive specifier.

The drive codes passed in DL are 0 = default, 1 = A, 2 = B, and so on.

Error return:

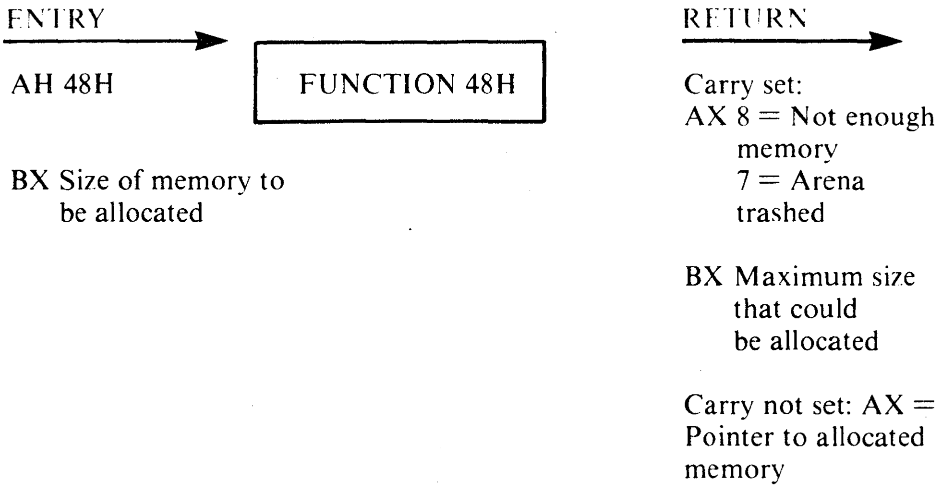
AX

15 = Invalid drive. The drive specified in DL was invalid.

Example:

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

ALLOCATE MEMORY



ENTRY

AH 48H

BX Size of memory to be allocated

FUNCTION 48H

RETURN

Carry set:
AX 8 = Not enough memory
7 = Arena trashed

BX Maximum size that could be allocated

Carry not set: AX = Pointer to allocated memory

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

Error returns:

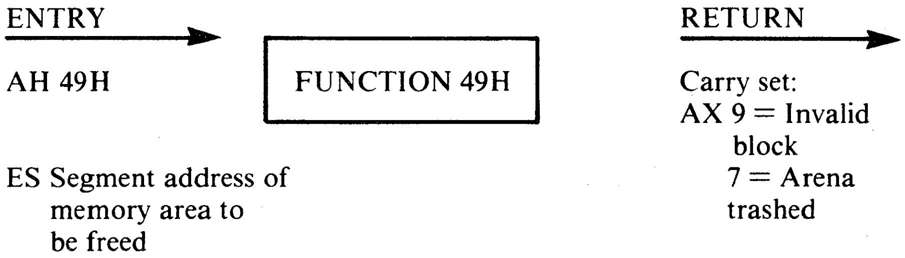
AX

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

Example:

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

FREE ALLOCATED MEMORY



ENTRY

AH 49H

ES Segment address of memory area to be freed

FUNCTION 49H

RETURN

Carry set:
AX 9 = Invalid block
7 = Arena trashed

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

Error returns:

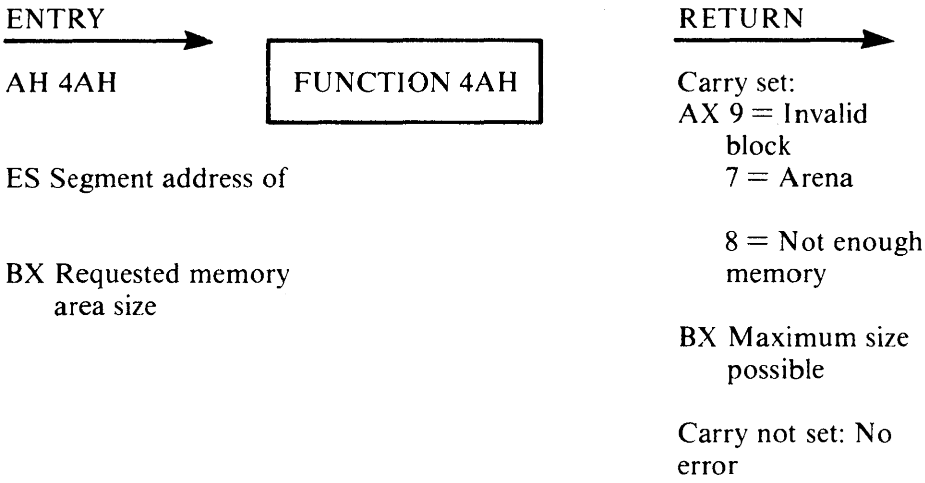
AX

9 = Invalid block. The block passed in ES is not one allocated via Function 48H.
7 = Arena trashed. The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it.

Example:

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

MODIFY ALLOCATED MEMORY BLOCKS



ENTRY

AH 4AH

ES Segment address of

BX Requested memory
area size

FUNCTION 4AH

RETURN

Carry set:
AX 9 = Invalid
block
7 = Arena

8 = Not enough
memory

BX Maximum size
possible

Carry not set: No
error

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

Error returns:

AX

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

Example:

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

LOAD AND EXECUTE A PROGRAM

ENTRY →

AH 4BH

DS:DX Pointer to pathname

ES:BX Pointer to parameter block

AL 00 = Load and execute program
03 = Load program

FUNCTION 4BH

RETURN →

Carry set:

function
10 = Bad environment
11 = Bad format
8 = Not enough
2 = File not found

Carry not set:
No error

Function 4BH allows a program to load another program into memory and begin execution of it (through a default).

DS:DX point to the ASCIZ name of the file to be loaded. ES:BX point to a parameter block for the load.

The following function codes are passed in AL.

AL	Function
0	Load and execute the program. A program header is established for the program and the terminate and CTRL-C addresses are set to the instruction after the EXEC system call.

NOTE

When control is returned, via a CTRL-C or terminate from the program being EXECed, all registers are altered including the stack. This is because control is returned from the EXECed program, not the system. To regain your stack, store an SS:SP value in a data location reachable from your CS.

3 Load (do not create) the program header and do not begin execution. This is useful in loading program overlays.

For AL = 0, the parameter block has the following format.

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

For AL = 3, the parameter block format is as follows.

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

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

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

(paragraph boundary)

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

Typically the environment strings have the format:

parameter = value

For example, COMMAND.COM always passes its execution search path as:

PATH = A:BIN;B:BASIC LIB

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

Note that on a successful return from EXEC, all registers, except for CS:IP, are changed.

Error returns:

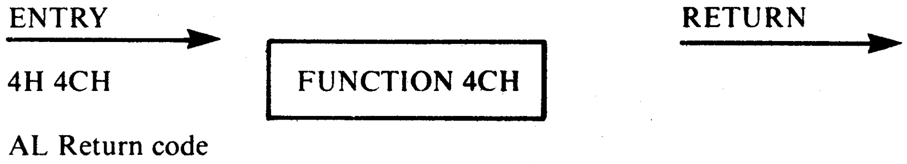
AX

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

Example:

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

TERMINATE A PROCESS



ENTRY

4H 4CH

AL Return code

FUNCTION 4CH

RETURN

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

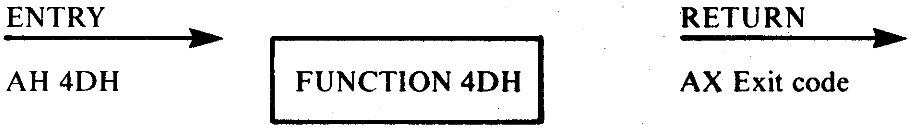
Error returns:

None.

Example:

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

RETRIEVE THE RETURN CODE OF A CHILD



ENTRY

AH 4DH

FUNCTION 4DH

RETURN

AX Exit code

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

Code	Function
0	Terminate/abort
1	CTRL-C
2	Hard error
3	Terminate and stay resident

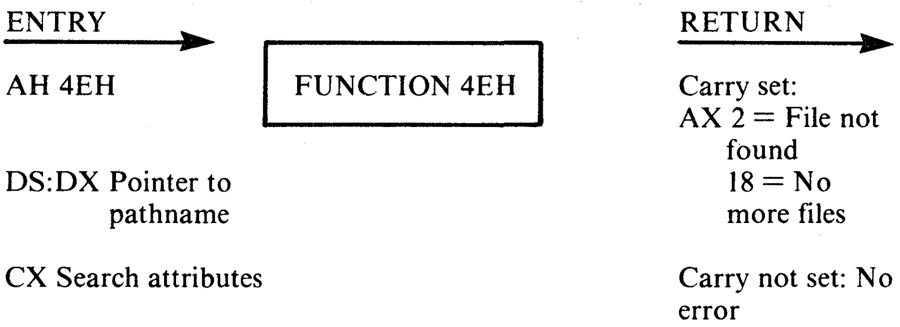
Error returns:

None.

Example:

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

FIND MATCH FILE



ENTRY

AH 4EH

DS:DX Pointer to pathname

CX Search attributes

FUNCTION 4EH

RETURN

Carry set:
AX 2 = File not found
18 = No more files

Carry not set: No error

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

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

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

Error returns:

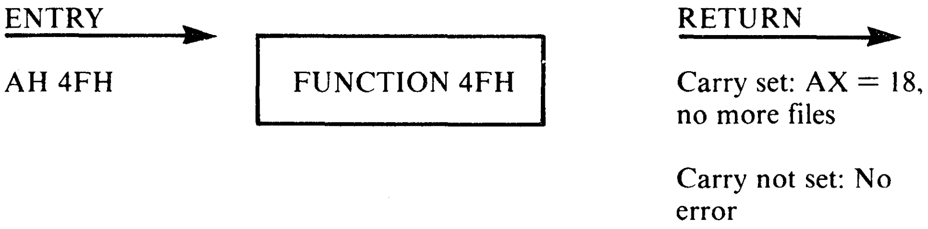
AX

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

Example:

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

STEP THROUGH A DIRECTORY MATCHING FILES



ENTRY

AH 4FH

FUNCTION 4FH

RETURN

Carry set: AX = 18, no more files

Carry not set: No error

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

Error return:

AX

18 = No more files. There are no more files matching this pattern.

Example:

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

RETURN CURRENT SETTING OF VERIFY AFTER WRITE FLAG

ENTRY →		RETURN →
AH 54H	FUNCTION 54H	AL Current verify flag value

Function 54H returns the current value of the verify flag in AL.

Error returns:

None.

Example:

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

MOVE A DIRECTORY ENTRY

ENTRY →		RETURN →
AH 56H	FUNCTION 56H	Carry set: AX 2 = File not found 17 = Not same device 5 = Access denied
DS:DX Pointer to pathname of existing file		
ES:DI Pointer to new pathname		Carry not set: No error

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

Error returns:

AX

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

Example:

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

GET/SET DATE/TIME OF A FILE

ENTRY →

AH 57H

AL 00 = get date and time
01 = set date and time

BX File Handle

CS If AL = 0, time to be set

DX If AL = 01, date to be set

FUNCTION 57H

RETURN →

Carry set:
AX 1 = Invalid function
6 = Invalid handle

Carry not set:
No error

CX:BX set if function 0

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

One of the following function codes is passed in AL.

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

Error returns:

AX

1 = Invalid function. The function passed in AL was not in the range 0:1.
6 = Invalid handle. The handle passed in BX was not currently open.

Example:

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

MACRO DEFINITIONS FOR MS-DOS SYSTEM CALL EXAMPLES

The following printout summarizes the Macro definitions used in the examples given for the MS-DOS system calls.

```
.xlist
;
;*******************
; Interrupts
;*******************
;
                                        ABS_DISK_READ
abs_disk_read macro disk,buffer,num_sectors,first_sector
              mov      al,disk
              mov      bx,offset buffer
```

```
           mov      cx,num_sectors
           mov      dx,first_sector
           int      37                       ;interrupt 37
           popf
           endm ;
;                                            ABS_DISK WRITE
abs_disk_write macro disk,buffer,num_sectors,first_sector
           mov      al,disk
           mov      bx,offset buffer
           mov      cx,num_sectors
           mov      dx,first_sector
int        38       ;interrupt 38
popf
endm
;
stay_resident macro last_instruc             ;STAY_RESIDENT
           mov      dx,offset last_instruc
           inc      dx
           int      39      .                ;interrupt 39
           endm
;
;******************
; Functions
;******************
;
read_kbd_and_echo macro                      ;READ_KBD_AND_ECHO
           mov      ah.1                     ;function 1
           int      33
           endm
;
display_char macro character                 ;DISPLAY_CHAR
           mov      dl.character
           mov      ah,2                     ;function 2
           int      33
           endm
;
aux_input  macro                             ;AUX_INPUT
           mov      ah,3                     ;function 3
           int      33
           endm
;
```

```
aux_output  macro                           ;AUX_OUTPUT

;;page
print_char macro    character               PRINT_CHAR
            mov     dl,character
            mov     ah,5                    ;function 5
            int     33
            endm
;
dir_console_input macro switch              ;DIR_CONSOLE_IO
            mov     dl,switch
            mov     ah,6                    ;function 6
            int     33
            endm
;
dir_console_input macro                     ;DIR_CONSOLE_INPUT
            mov     ah,7                    ;function 7
            int     33
            endm ;
read_kbd    macro                           ;READ_KBD
            mov     ah,8                    ;function 8
            int     33
            endm ;
display     macro   string                  ;DISPLAY
            mov     dx,offset string        ;function 9
            mov     ah,9
            int     33
            endm
;
get_string macro    limit,string            ;GET STRING
            mov     string,limit
            mov     dx,offset string
            mov     ah,10                   ;function 10
            int     33
            endm
;
check_kbd_status macro                      ;CHECK_KBD_STATUS
            mov     ah,11                   ;function 11
            int     33
            endm
```

```
;
flush_and_read_kbd  macro switch              ;FLUSH_AND_READ_KBD
            mov     ai,switch
            mov     ah,12                     ;function 12
            int     33
            endm
;
reset_disk_macro         ;RESET DISK
            mov     ah,13                     ;function 13
            int     33
            endm


;;page
select_disk macro   disk                      ;SELECT_DISK

            mov     dl,disk¿-65¡
            mov     ah,14                     ;function 14
            int     33
            endm
;
open        macro   fcb                       ;OPEN
            mov     dx,offset fcb
            mov     ah,15                     ;function 15
            int     33
            endm
;
close       macro   fcb                       ;CLOSE
            mov     dx,offset fcb
            mov     ah,16                     ;function 16
            int     33
            endm
;
search_first macro  fcb                       ;SEARCH_FIRST
            mov     dx,offset fcb
            mov     ah,17                     ;Function 17
            int     33
            endm
```

```
;
search_next macro   fcb                     ;SEARCH_NEXT
            mov     dx,offset fcb
            mov     ah,17                   ;function 17
            int     33
            endm
;
delete      macro   fcb                     ;DELETE
            mov     ah,18                   ;function 19
            int     33
            endm
;
read_seq    macro   fcb                     ;READ_SEQ
            mov     dx,offset fcb
            mov     ah,20                   ;function 20
            int     33
            endm
;
write_seq   macro   fcb                     ;WRITE_SEQ
            mov     dx,offset fcb
            mov     ah,21                   ;function 21
            int     33
            endm
;
create      macro   fcb                     ;CREATE
            mov     dx,offset fcb
            mov     ah,22                   ;function 22
            int     33
            endm
;

rename      macro   fcb,newname             ;RENAME
            mov     dx,offset fcb
            mov     ah,23                   ;function 23
            int     33
            endm
;
current_disk macro                          ;CURRENT_DISK
            mov     ah,25                   ;function 25
            int     33
            endm
```

```
;
set_dta     macro   buffer                    ;SET_DTA
            mov     dx,offset buffer
            mov     ah,26                     ;function 26
            int     33
            endm
;
alloc_table macro                             ;ALLOC_TABLE
            mov     ah,27                     ;function 27
            int     33
            endm
;
read_ran    macro   fcb                       ;READ_RAN
            mov     dx,offset fcb
            mov     ah,33                     ;function 33
            int     33
            endm
;
write_ran   macro   fcb                       ;WRITE_RAN
            mov     dx,offset fcb
            mov     ah,34                     ;function 34
            int     33
            endm
;
file_size   macro   fcb                       ;FILE_SIZE
            mov     dx,offset fcb
            mov     ah,35                     ;function 35
            int     33
            endm
;
set_relative_record macro fcb                 ;SET_RELATIVE_RECORD
            mov     dx,offset fcb
            mov     ah,36                     ;function 36
            int     33
            endm
;;page
set_vector  macro interrupt,seg_addr,off_addr ;SET_VECTOR
            push    ds
            mov     ax,seg_addr
            mov     ds,ax
            mov     dx,off_addr
```

```
          mov     al,interrupt
          mov     ah,37                     ;function 37
          int     33
          endm

;
create_prog_seg  macro  seg_addr             ;CREATE_PROG_SEG
          mov     dx,seg_addr
          mov     ah,38                     ;function 38
          int     33
          endm
;
ran_block_read macro fcb,count,rec_size      ;RAN_BLOCK_READ
          mov     dx,offset fcb
          mov     cx,count
          mov     word ptr fcb[14],rec_size
          mov     ah,39                     ;function 39
          int     33
          endm
;
ran_block_write macro fcb, count, rec_size   ;RAN_BLOCK_WRITE
          mov     dx,offset fcb
          mov     cx,count
          mov     word ptr fcb[14],rec_size
          mov     ah,40                     ;function 40
          int     33
          endm
;
parse     macro   filename,fcb              ;PARSE
          mov     si,offset filename
          mov     di,offset fcb
          push    es
          push    ds
          pop     es
          mov     al,15
          mov     ah,41                     ;function 41
          int     33
          pop     es
          endm
```

```
;
get_date    macro   ;GET DATE
            mov     ah,42                   ;function 42
            int     33
            endm
;;page
set_date    macro   year,month,day          ;SET_DATE
            mov     cx,year
            mov     dh,month
            mov     dl,day
            mov     ah,43                   ;function 43
            int     33
            endm
;
get_time    macro   ;GET_TIME
            mov     ah,44                   ;function 44
            int     33
            endm
;
                                            ;SET_TIME
set_time    macro   hour,minutes,seconds,hundredths
            mov     ch,hour
            mov     cl,minutes
            mov     dh,seconds
            mov     dl,hundredths
            mov     ah,45                   ;function 45
            int     33
            endm
;
verify      macro   switch                  ;VERIFY
            mov     al,switch
            mov     ah,46                   ;function 46
            int     33
            endm
;
;*******************
; General
;*******************
;
move_string macro source,destination,num_bytes
                                            ; MOVE_STRING
```

```
                push    es
                mov     ax,ds
                mov     cs,ax
                assume  es:data
                mov     si,offset source
                mov     di,offset destination
                mov     cx,num_bytes
        rep     movs    es:destination,source
                assume  es:nothing
                pop     es
                endm
;
;
convert         macro   value,base,destination          ;CONVERT
                local   table,start
                jmp     start
table           db      "0123456789ABCDEF"
start:          mov     al,value
                xor     ah,ah
                xor     bx,bx
                div     base
                mov     bl,al
                mov     al,cs:table[bx]
                mov     destination,al
                mov     bl,ah
                mov     al,cs:table[bx]
                mov     destination[1],al
                endm
;;page
convert_to_binary          macro string,number,value
                                                ;CONVERT_TO_BINARY
                local   ten, start,calc,mult,no_mult
                jmp     start
ten             db      10
start:          mov     value,0
                xor     cx,cx
                mov     cl,number
calc:           xor     ax,ax
                mov     al,string [si]
                sub     al,48
                cmp     cx,2
```

```
            jl      no_mult
            push    cx
            dec     cx
mult:       mul     cs:ten
            loop    mult
            pop     cx
no_mult:    add     value,ax
            inc     si
            loop    calc
            endm
;
convert_date macro      dir_entry
            mov     dx,word ptr dir_entry[25]
            mov     cl,5
            shr     dl,cl
            mov     dh,dir_entry [25]
            and     dh,lfh
            xor     cx,cx
            mov     cl,dir_entry[26]
            shr     cl,l
            add     cx,1980
            endm
;
```

AN EXTENDED EXAMPLE OF MS-DOS SYSTEM CALLS

The following program provides more examples of system calls.

```
title DISK DUMP
zero                equ             0
disk_B              equ             1
sectors_per_read    equ             9
cr                  equ             13
blank               equ             32
period              equ             46
tilde               equ             126
          INCLUDE B:CALLS.EQU
;
subttl DATA SEGMENT
page +
data                                segment
;
```

```
input_buffer                    db        9 dup(512 dup(?))
output_buffer                   db        77 dup(" ")
                                db        0DH,0AH,"$"
start_prompt                    db        "Start at sector: $"
sectors_prompt                  db        "Number of sectors: $"
continue_prompt                 db        "RETURN to continue $"
header                          db        "Relative sector $"
end_string                      db        0DH,0AH,0AH,07H,"ALL
                                          DONE$"
                                          ;DELETE THIS
crlf                            db        0DH,0AH,"$"
table                           db        "0123456789ABCDEF$"
;
ten                             db        10
sixteen                         db        16
;
start_sector                    dw        1
sector_num    label             byte
sector_number                   dw        0
sectors_to_dump                 dw        sectors_per_read
sectors_read                    dw        0
;
buffer        label             byte
max_length                      db        0
current_length                  db        0
digits                          db        5 dup(?)
;
data                            ends
;
subttl STACK SEGMENT
page +
stack                           segment             stack
                                dw        100 dup(?)
stack_top                       label     word
stack                           ends
;
subttl MACROS
page +
;
```

```
           INCLUDE B:CALLS.MAC
;BLANK LINE

blank line              macro                   number
                        local                   print_it
                        push                    cx
                        call                    clear_line
                        mov                     cx,number
print_it:               display                 output_buffer
                        loop                    print_it
                        pop                     cx
                        endm
;
subttl ADDRESSABILITY
page +
code                    segment
                        assume                  cs:code,ds:data,ss:stack
start:                  mov                     ax,data
mov                     ds,ax
mov                     ax,stack
mov                     ss,ax
mov                     sp,offset stack_top
;
                        jmp                     main_procedure
subttl PROCEDURES
page +
;
; PROCEDURES
; READ_DISK
read_disk               proc;
                        cmp                     sectors_to_dump,zero
                        jle                     done
mov                     bx,offset input_buffer
                        mov                     dx,start_sector
                        mov                     al,disk_b
                        mov                     cx, sectors_per_read
                        cmp                     cx, sectors_to_dump
                        jle                     get_sector
                        mov                     cx, sectors_to_dump
get_sector:             push                    cx
                        int                     disk_read
```

```
                    popf
                    pop                 cx
                    sub                 sectors_to_dump,cx
                    add                 start_sector,cx
                    mov                 sectors_read,cx
xor                 si,si
done:               ret
read_disk           endp
;CLEAR LINE

clear_line          proc;
                    push                cx
                    mov                 cx,77
                    xor                 bx,bx
move_blank:         mov                 output_buffer[bx],' '
                    inc                 bx
                    loop                move_blank
                    pop                 cx
                    ret
clear_line          endp
;
;PUT_BLANK
put_blank           proc;
                    mov                 output_buffer [di]," "
                    inc                 di
                    ret
put_blank           endp
;
;
setup               proc;
                    display             start_prompt
                    get_string          4,buffer
                    display             crlf
convert_to_binary digits,
current_length,start_sector
mov                 ax,start_sector
                    mov                 sector_number,ax
                    display             sectors_prompt
                    get_string          4,buffer
                    convert_to_binary digits,
                        current_length,sectors_to_dump
                        ret
setup                   endp
;
;CONVERT_LINE
convert_line            proc;
                        push            cx
                        mov             di,9
                        mov             cx,16
convert_it:             convert         input_buffer [si],sixteen,
                        output_buffer [di]
                        inc             si
                        add             di,2

                        call            put_blank
                        loop            convert_it
                        sub             si,16
                        mov             cx,16
                        add             di,4
display_ascii:          mov             output_buffer [di],period
                        cmp             input_buffer [si],blank
                        jl              non_printable
                        cmp             input_buffer[si],tilde
                        jg              non_printable
printable:              mov             dl,input_buffer [si]
                        mov             output_buffer [di],dl
non_printable:          inc             si
                        inc             di
                        ioop            display_ascii
                        pop             cx
                        ret
convert_line            endp
;
display_screen          proc;
                        push            cx
                        call            clear_line
;
                        mov             cx,17
;I WANT length header
dec                     cx
;minus 1 in cx
```

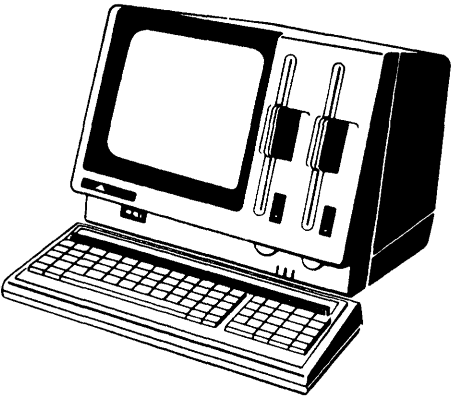
```
                    xor                     di,di
move_header:        mov                     al,header [di]
                    mov                     output_buffer  [di],al
                    inc                     di
                    loop                    move_header ;FIX THIS!
,
                    convert                 sector_num[1],sixteen,
                    output_buffer[di]
                    add                     di,2
                    convert                 sector_num,sixteen,
                    output_buffer [di]
                                                                      display
                    blank_line 2
                                                                      mov
dump_it:            call                    clear_line

                    call                    convert_line
                    display                 output_buffer
                    loop                    dump_it
                    blank_line 3
                    display                 continue_prompt
                    get_char_no_echo
                    display                 crlf
                    pop                     cx
                    ret                     display_screen endp
;
;
; END PROCEDURES
subttl MAIN PROCEDURE
page +
main_procedure:     call                    setup
check_done:         cmp                     sectors_to_dump,zero
                    jng                     all_done
                    call                    read_disk
                    mov                     cx,sectors_read
display it:         call                    display_screen
                    call                    display_screen
                    inc                     sector_number
                    loop                    display_it
                    jmp                     check_done
```

```
all_done:       display         end_string
                get_char_no_echo
                ends
code            ends
                end     start
```

Chapter 3

The Extended I/O System Functions



Calls to extended I/O System functions from user programs are issued directly to IO.SYS, bypassing MSDOS.SYS.

Entry to these functions is accomplished through the software interrupt 220H. Extended function calls use registers for passing function codes and parameters.

- Register CL holds the function code.
- Registers DX, DS, and AX contain additional parameters as necessary.

All registers are automatically saved upon entry and restored upon exit from the extended function call.

GET TIME AND DATE

ENTRY →		RETURN →
CL 00H	EXT FUNC 00H	Buffer Time and date
DS:DX Data Buffer Address		

Extended Function 00H returns the system time and date. Registers DS and DX hold the address of the I/O data buffer in which the data is to be stored. The system fills the data buffer at the indicated address in the following format.

Year
Month Day of Week*
Day
Hour
Minute
Second

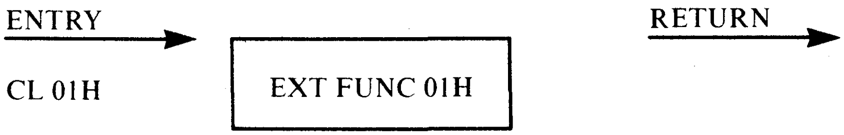
<----------1 byte--------->

*Month and Day of Week are each half byte-values.

Year=00-99	BDC	Day=1-31	BCD
Month=1-12	Hex	Hour=0-23	BCD
Day of Week=1-7	Hex	Minute=0-59	BCD
(1=Sun. 2=Mon.,		Second=0-59	BCD
and so on)			

The Get Time and Date extended function performs the same operations as the Get Time and Get Date function requests (Functions 2CH and 2AH).

SET TIME AND DATE



ENTRY

RETURN

CL 01H

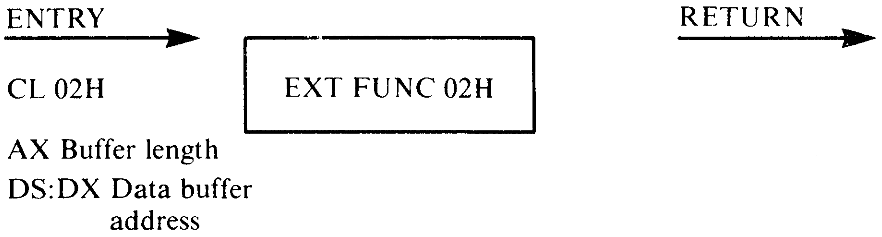
EXT FUNC 01H

DS:DX Data Buffer
Address

Extended Function 00H sets the system time and date. The buffer addressed by registers DS and DX must contain the time and date. The I/O data buffer format is the same as that used by Extended Function 00H, Get Time and Date.

The Set Time and Date extended function performs the same operations as the Set Time and Set Date function requests (Functions 2DH and 2BH).

PLAY MUSIC



CL 02H

AX Buffer length

DS:DX Data buffer address

Extended Function 02H plays music on the APC. The I/O buffer addressed by registers DS and DX consists of melody data. Register AX is set to the I/O buffer length in bytes.

Melody data consists of two types of information: control commands and scale data. Control commands set the loudness and speed. Scale data refer to notes, duration, and accent.

Control Data

Control data is written in the following format:

[M[n]] [T[n]]

Table 3-1 lists the acceptable values for n. Both the loudness and speed commands are optional, as indicated by the square brackets. The values are effective until new ones are specified.

Table 3-1 Melody Data Control Commands

COMMAND	FUNCTION
Mn	Loudness n = 1 piano 2 medium (default) 3 forte
Tn	Speed n = 1 1.00 sec for quarter note 2 0.87 sec (default) 3 0.56 sec 4 0.38 sec

Scale Data

Scale data sets the note values, duration, and accent. The allowable values for these variables are defined in Tables 3-2 and 3-3.

Table 3-2 Note Values

NOTE	FUNCTION
-C	low octave
-C#	
-D	
-D#	
-E	
-F	
-F#	
-G	
-G#	
-A	
-A#	
-B	
C	middle octave
C#	
D	
D#	
E	
F	
G	
G#	
A	
A#	
B	
+C	high octave
+C#	
+D	
+D#	
+E	
N	rest

Table 3-3 Duration Values

DURATION	FUNCTION (FOR REST NOTE)
0	𝅝 whole 𝄻
1	𝅗𝅥. dotted 1/2 𝄼.
2	𝅗𝅥 1/2 𝄼
3	𝅘𝅥. dotted 1/4 𝄽.
4	𝅘𝅥 1/4 𝄽
5	𝅘𝅥𝅮. dotted 1/8 𝄾.
6	𝅘𝅥𝅮 1/8 𝄾
7	𝅘𝅥𝅯. dotted 1/16 𝄿.
8	𝅘𝅥𝅯 1/16 𝄿
9	𝅘𝅥𝅰 1/32 𝅀

The format of the scale data command is as follows:

[S] note [duration]

The accent command is indicated by the value S in the scale data command. Both accent and duration are optional. The accent applies only to the note value it precedes. The duration is effective until the next duration is specified.

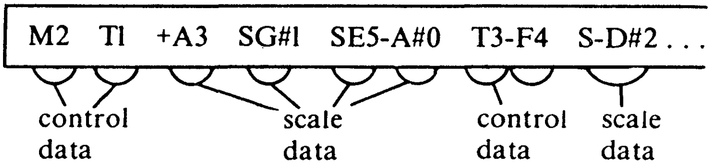
Complete Melody Data Format

The complete melody data format, then, is

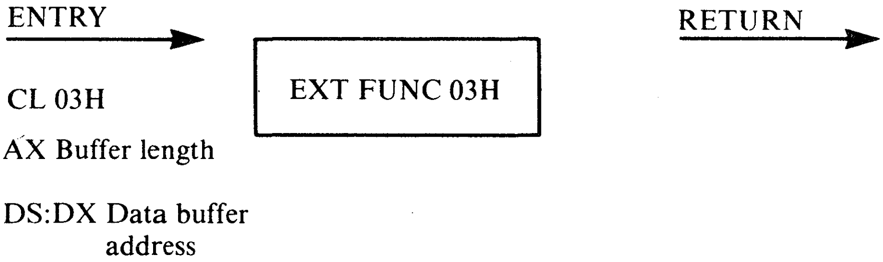
[control data] [scale data] ...

The control data is effective until the next control data is specified.

An example of melody data follows.



SOUND BEEP


ENTRY

CL 03H

AX Buffer length

DS:DX Data buffer address

EXT FUNC 03H

RETURN

Extended Function 03H sounds the beep tone on the APC. The I/O buffer addressed by registers DS and DX contains beep data. Register AX is set to the I/O buffer length in bytes.

Beep data consists of control commands and parameters. Control commands set the loudness and type of sound. The parameters control frequency and tone period.

Control Commands

Control commands are written in the following format.

$$\left[\begin{Bmatrix}\mathrm{B}\\\mathrm{P}\end{Bmatrix}\ [\mathrm{n}]\right]$$

The loudness parameter, n, is optional. Table 3-4 lists the values for n. Control data is effective until the next control data is specified. B and P are mutually exclusive commands; they cannot be specified together.

Table 3-4 Short Sound Control Commands

COMMAND	FUNCTION
Bn	B = Rectangular wave sound (beep)
Pn	P = Piano sound
	n = Loudness 1 piano 2 medium (default) 3 forte

The parameter format is a frequency value followed, optionally, by a number specifying the tone period.

$$\begin{Bmatrix}\mathrm{H}\\\mathrm{I}\\\mathrm{J}\\\mathrm{K}\end{Bmatrix}[\mathrm{n}]$$

Beep Sound Parameters

The beep sound parameters and their corresponding values are defined in Table 3-5.

Table 3-5 Beep Sound Parameters

PARAMETER	VALUE	MEANING
Frequency	H	710 Hz
	I	1202 Hz
	J	2038 Hz
	K	3406 Hz
Tone period	1	20 msec (min)
n	2	2x10 msec
	3	3x10 msec
	.	
	.	
	.	
	N	Nx10 msec
	.	
	.	
	.	
	65535	65535x10 msec

Complete Beep Command Format

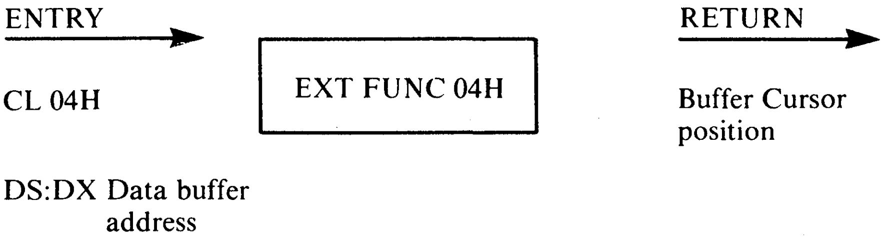
The complete format of the beep command is

[control data] [sound parameter]...

Both parts of the command are optional. An example of a command follows.

```
P2  K8  B1  H3. . .
```

REPORT CURSOR POSITION ENTRY



ENTRY

CL 04H

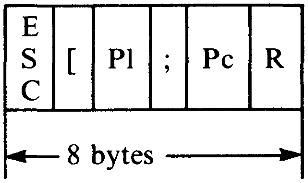
DS:DX Data buffer address

EXT FUNC 04H

RETURN

Buffer Cursor position

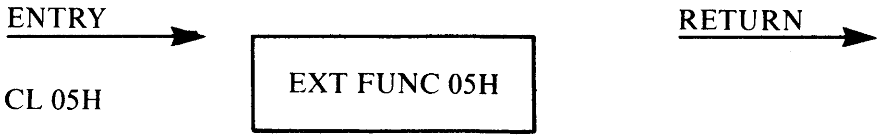
Extended Function 04H gets the current active position on the console screen. Registers DS and DX point to the address of the I/O buffer in which the data is to be stored. The system returns the column and line numbers of the current position prefixed by the escape (ESC) code in the following format:



ESC	[	Pl	;	Pc	R

All characters are returned as ASCII code values. P1 is the line number (01-25). Pc is the column number (01-80).

AUTO POWER OFF



ENTRY

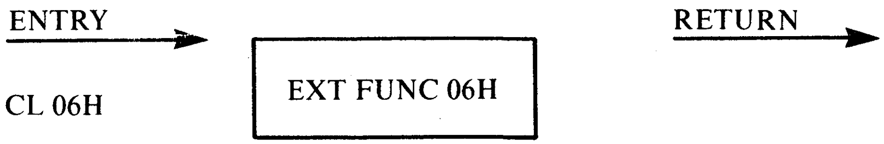
CL 05H

EXT FUNC 05H

RETURN

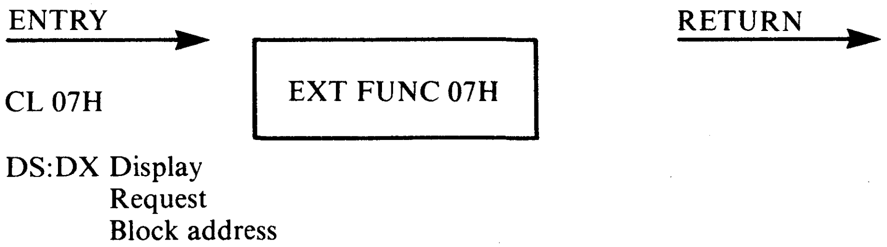
Extended Function 05H turns off the power of the APC. When this function is called, the system waits approximately five seconds before turning off the power. To turn the system back on, turn the APC power switch off, then turn it back on.

INITIALIZE KEYBOARD FIFO BUFFER



Extended Function 06H initializes the keyboard FIFO buffer. This function does not pass any parameters.

DIRECT CRT I/O



Extended Function 07H allows the assembly language programmer to perform high speed block level I/O operations to the console through the DMA. Five different operations may be performed through this function. They are identified by the command number passed in the Display Request Block. The Extended Function 07H commands are listed in Table 3-6.

Table 3-6 Direct CRT I/O Function Commands

CMD#	FUNCTION
0	Display video memory format data on CRT
1	Display string data on CRT
2	Report cursor position by binary value
3	Roll down screen
4	Roll up screen

Figure 3-1 shows how the DMA transfer function works. The Display Request Block contains the addresses of display data in video memory format, and attribute data. This data is transferred to the Display Data Area and the Attribute Data Area, respectively, in video memory.

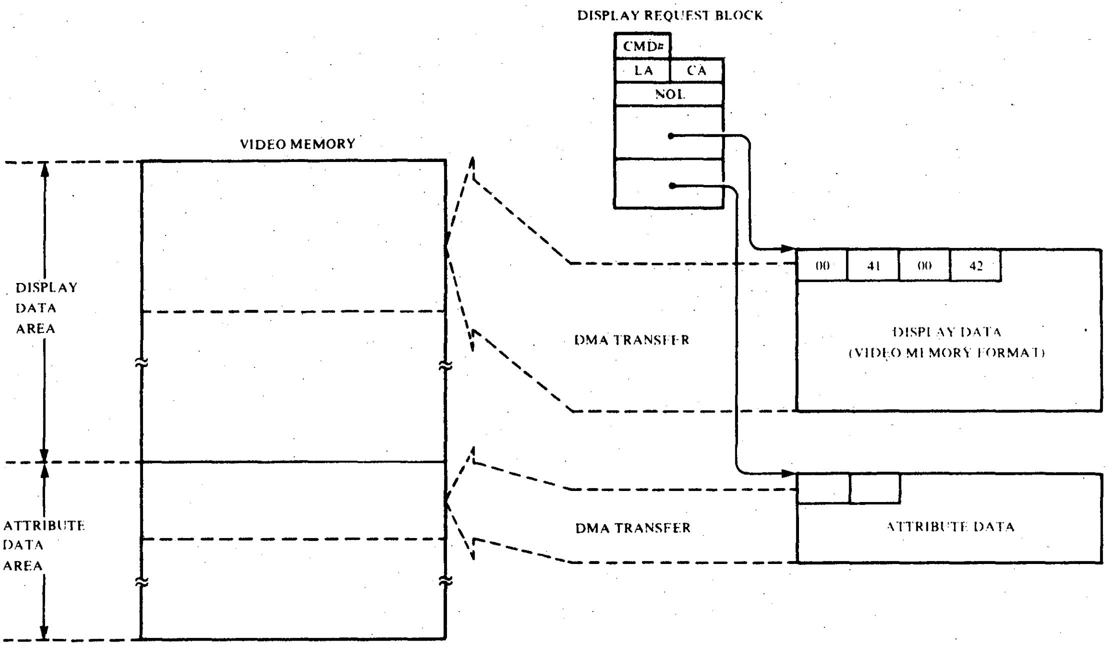


Figure 3-1 DMA Transfer

Display Request Block

The Display Request Block used in the Direct CRT I/O function contains control data for the DMA exchange. It includes the command number, cursor position from which the data is to be displayed, the number of characters to display, and the address of the data buffer. Registers DS and DX are set to the address of the Display Request Block prior to issuing the function call. The format of the Display Request Block is shown below.

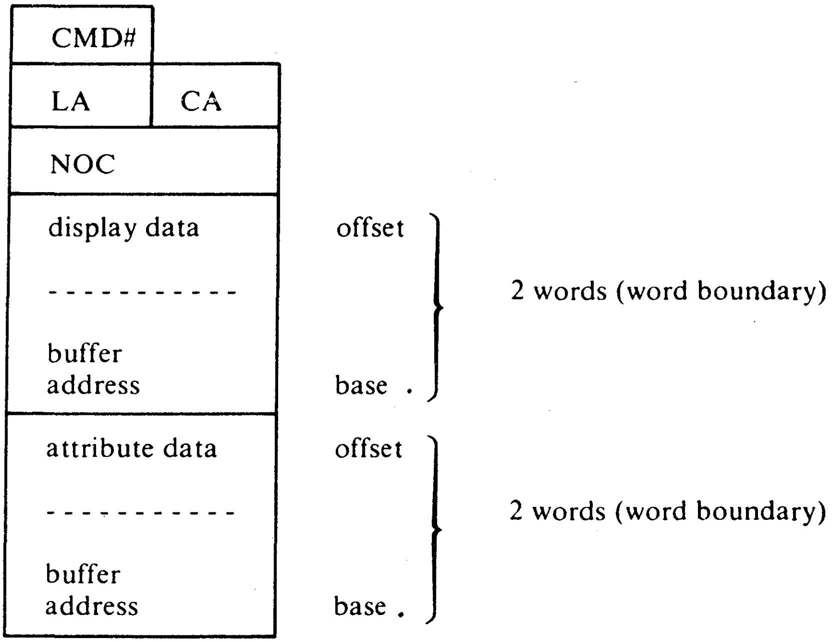
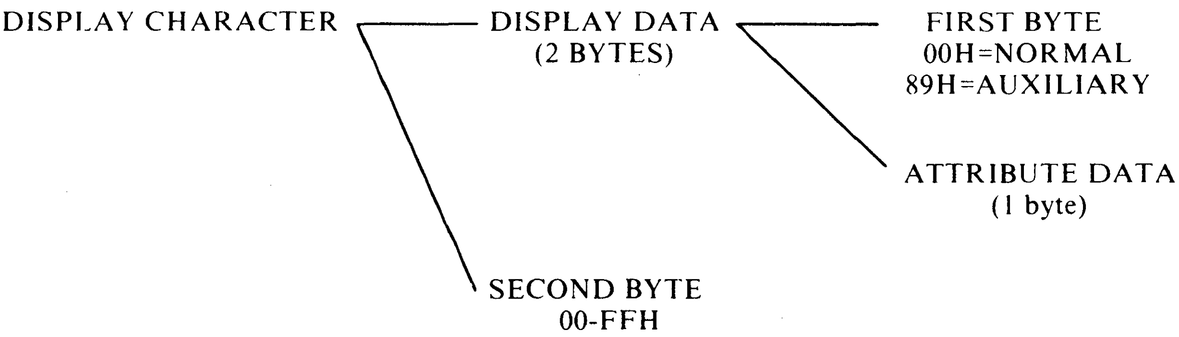


Figure 3-2 Display Request Block

The data fields in the Display Request Block are the following:

CMD#:	0 - 4 (Command Number)
LA.CA:	Display/cursor position LA (Line address) = 0-24 binary, 1 byte CA (Column address) = 0-79 binary, 1 byte
NOC:	Number of characters to be displayed 0-2000 binary, 1 word
Display data address:	Starting address of display data buffer (offset, base address; 2 words)
Attribute data address:	Starting address of attribute data buffer (offset, base address; 2 words)

In the video memory, each display character consists of display data (two bytes) and attribute data (one byte). The first byte of the display data identifies whether the next code is the normal character code or the auxiliary character code, as shown in the following illustration.



With CMD#0, both normal and auxiliary character codes may be used in the video memory format. With CMD#1, only normal character codes may be used.

Video Memory Format

Video memory format is the format of the Display Data Area in the video memory. Each display data item consists of two bytes.

display data 1		display data 2		display data 3		. . .
first byte	second byte	first byte	second byte	first byte	second byte	. . .

first byte = 00H (normal character code)
89H (auxiliary character code)

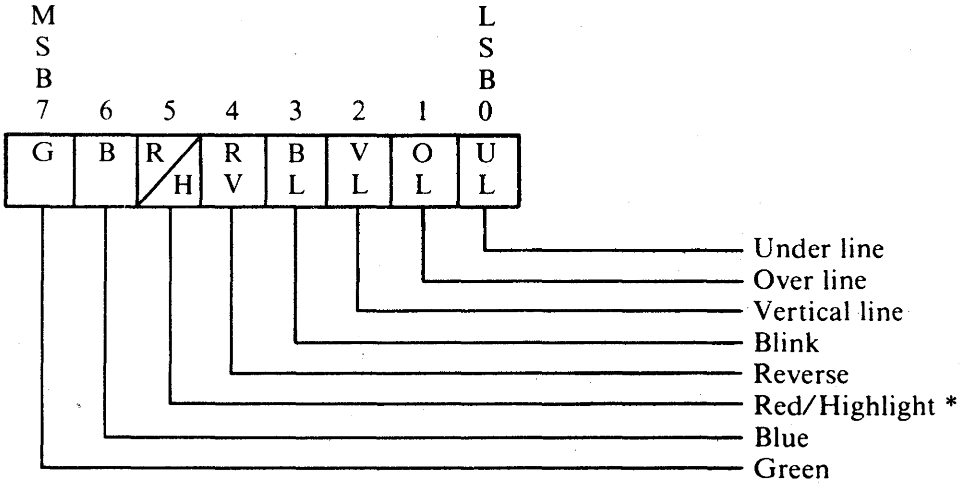
second byte = 00H - FFH (normal or auxiliary character code)

String Data Format

In the string data format for CMD#1, each display data item is one byte long, and only normal character codes are available.

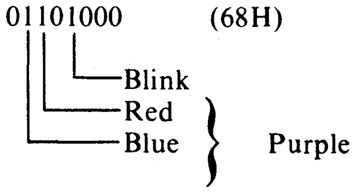
Attribute Data Format

The attribute data items occur in one-to-one correspondence with the display data items. That is, there is one attribute data item for each display data item. Each attribute data item is one byte in length, with each of the eight low-order bits set to 0 or 1 to indicate no color or a color value. The colors are assigned to bits as follows.



* - Highlight is available for monochrome monitor only.

Colors may be used individually or in combination to generate secondary colors. For example, the following attribute data byte displays data with blink and purple color attributes.



Direct CRT I/O Command Descriptions

CMD# 0 - DISPLAY VIDEO MEMORY FORMAT DATA ON CRT

This function displays the data, starting from the positions specified by LA and CA for the length in NOC, on the CRT. The display data must be formatted in the video memory format.

The contents of the display request block for this command follow.

LA   Range is 0-24, binary. Values greater than 24 are converted to 24.

CA   Range is 0-79, binary. Values greater than 79 are converted to 79.

NOC   If the number of data items to be displayed exceeds the display area on the CRT, the overflow data is ignored. If NOC is 0, the cursor is positioned at LA and CA, and no other action is taken.

Display data address   The starting address should be located at an even memory address (DMA controller's restriction). If the base address is 0, no display data is transferred.

Attribute data address   If the base address is 0, attribute data is not transferred.

If the base addresses of both display data and attribute data are 0, the effect is the same as setting NOC to 0. The cursor is positioned at LA,CA and no data is transferred.

After data is transferred, the cursor is positioned at the next cursor position. If the cursor is positioned on the last screen position (25,80) when the call is issued, the command is executed, the screen rolls up one line, and the cursor is positioned on the first field of the bottom line.

CMD# 1 - DISPLAY STRING DATA ON CRT

This command, like CMD# 0, displays the data addressed by LA and CA for the length in NOC on the CRT. The display data must be in string data format with each item consisting of one byte of normal character code data.

The contents of the Display Request Block are the same for this command as for CMD# 0, except that CMD# is 1.

CMD# 2 - REPORT CURSOR POSITION

This command returns the current cursor position in fields LA and CA in the Display Request Block. The function uses only the Display Request Block fields listed below. The contents of the remainder of the area are ignored.

LA Line address (0-24, binary)

CA Column address (0-79, binary)

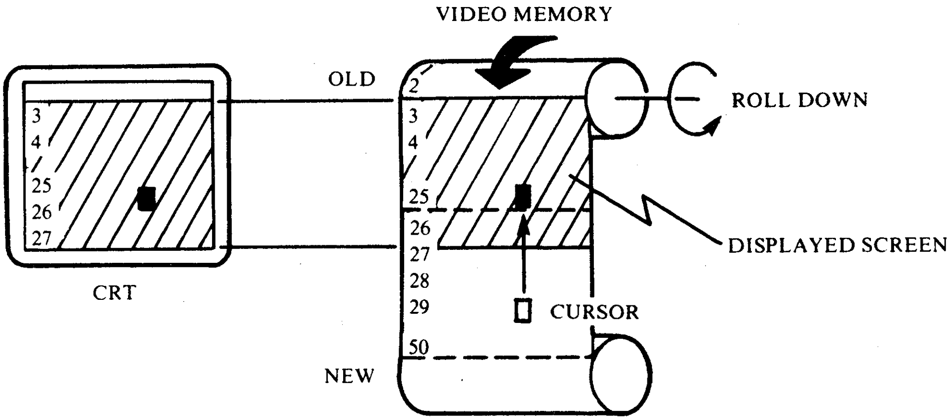
CMD# 3 - ROLL DOWN SCREEN

This command enables the programmer to roll down a maximum of 25 lines on the screen. The function uses only the LA field in the Display Request Block. The contents of the remainder of the area are ignored.

LA Number of lines to roll down (1-25, binary)

The following illustrates the roll down operation.

LA Number of lines to roll down (1-25, binary)

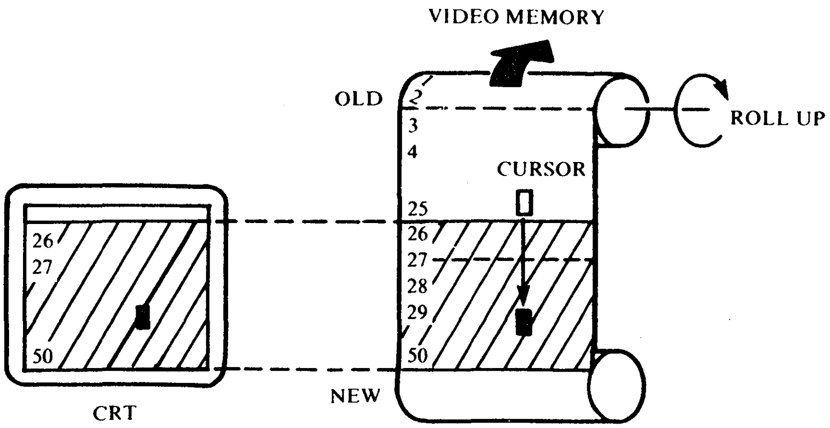


CMD# 4 - ROLL UP SCREEN

This command enables the programmer to roll up a specified number of lines on the screen. The function uses only the LA field in the Display Request Block. The contents of the remainder of the area are ignored.

LA Number of lines to roll up. If the number of lines to roll up exceeds the number of lines that have been written, the next line is erased.

The following illustrates the roll up operation.



WRITE CMOS

ENTRY →

EXT FUNC 08H

RETURN →

CL 08H

DS:DX Data buffer
address

Extended 08H writes up to 512 bytes to CMOS RAM (battery back-up memory). The data to be written is stored in an I/O buffer addressed by registers DS and DS. The format of the buffer is as follows.

READ CMOS

ENTRY →

EXT FUNC 09H

RETURN →

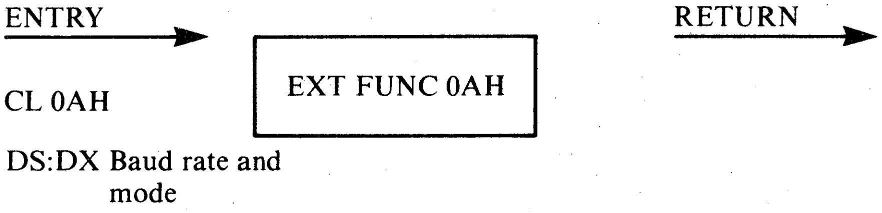
CL 09H

Data buffer

DS:DX Data buffer
address

Extended Function 09H reads data in CMOS RAM (battery back-up memory) into the buffer addressed by registers DS and DX. The system fills the data buffer in the format defined in Function 08H, Write CMOS.

INITIALIZE RS-232C



ENTRY

CL 0AH

DS:DX Baud rate and mode

EXT FUNC 0AH

RETURN

The Initialize RS-232C function is used in asynchronous mode only to set the baud rate (DH) and mode (DL). (In synchronous mode, an external clock determines the baud rate.) The register values are set as follows.

DH = Baud Rate

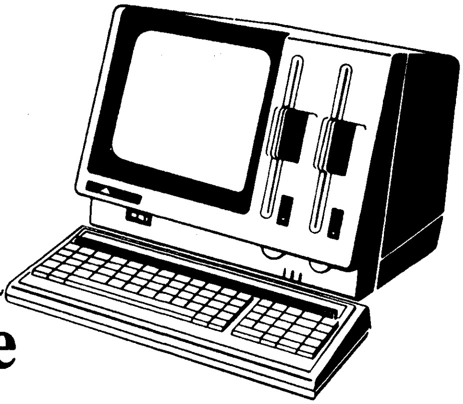
0 =	150 BPS
1 =	200 BPS
2 =	300 BPS
3 =	600 BPS
4 =	1200 BPS
5 =	2400 BPS
6 =	4800 BPS
7 =	9600 BPS
8 =	19200 BPS

DL = Asynchronous mode byte for PD8251

An illustration of the control information, including baud rate, for data transmission follows.

NOTE

When communication software is operating, the system timer is off and the keyboard repeat feature does not operate.



Chapter 4

The APC Escape Sequence Functions

When a program calls an APC escape sequence function, it uses the following function requests:

- Function request 02H (Console Output)
- Function request 06H (Direct Console I/O).

ESCAPE SEQUENCE FORMAT

Escape sequences consist of three fields: a sequence introducer that identifies the instruction as an escape sequence, one or more parameters, and a final character. For example, the format of the escape sequence to move the cursor up is

```
ESC [PnA
```

The basic elements of all APC escape sequences are the same.

- The Control Sequence Introducer (CSI) signals an escape sequence command to the system. For the APC, the CSI is the ESC character (1BH).

  The ESC is usually, but not always, followed by a square bracket ([).

- A parameter is a string of zero or more decimal characters that represent a single value. Leading zeroes are ignored. The decimal characters have a range of 0 (30H) to 9 (39H). Two types of parameters are used in escape sequences: numeric and selective parameters.

  Numeric parameters represent numbers. Unless otherwise specified, any numeric value may be used. Numeric parameters are designated Pn in this document.

Selective parameters, designated in this document by Ps, select a subfunction from a specified list of subfunctions.

You must replace Pn and Ps as well as certain command-specific parameters with the appropriate values in the command.

A parameter string is a list of parameters, separated by semicolons (3BH).

A default is a function-dependent value that is assumed when no value is explicitly specified for a parameter.

- The Final Character is a character whose bit combination terminates an escape or control sequence. There is a different character for each escape sequence. In the example above, "A" is the Final Character. The Final Character must be entered exactly as it appears in the command format. Be careful to use uppercase or lowercase correctly.

For example, the following escape sequence sets character attributes.

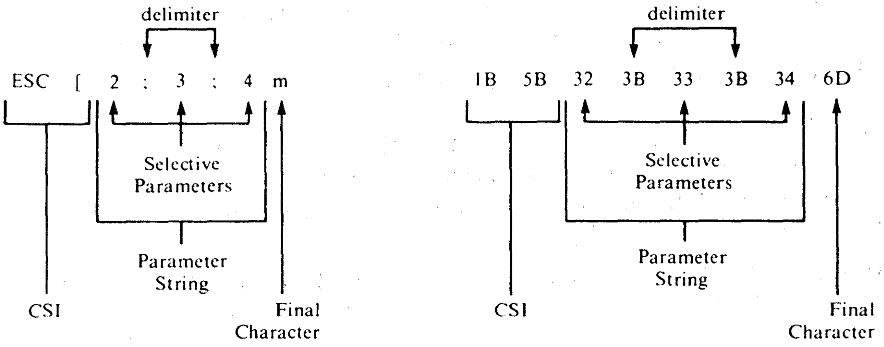
ESC [Ps;...; Psm

To select the attributes "over line" (3), "under line" (4), and "blink" (5), you would enter the values that correspond to the following sequence. All the character attributes for display are listed in Table 4-1.

ESC [3;4;5m

Note that lowercase m is used in this command as the final character.

The escape sequence is represented below in both decimal and hexadecimal values.



CURSOR UP

ESC[PnA Default value: 1

This sequence moves the active position up without altering the column position. The number of lines moved is determined by the parameter. A parameter value of 0 or 1 moves the active position up one line. A parameter value of n moves the active position up n lines. If an attempt is made to move the cursor above the first character of the first display line, the cursor stops at the top margin.

CURSOR DOWN

ESC[PnB Default value: 1

This sequence moves the active position down without altering the column position. The number of lines moved is determined by the parameter. A parameter value of 0 or 1 moves the active position down one line. A parameter value of n moves the active position down n lines. If an attempt is made to move the cursor below the bottom margin, the screen rolls up the required number of lines.

CURSOR FORWARD

ESC[PnC Default value: 1

This sequence moves the active position to the right. The distance moved is determined by the parameter. A parameter value of 0 or 1 moves the active position one position to the right. A parameter value of n moves the active position n positions to the right. If an attempt is made to move the cursor to the right of the right margin, the cursor moves to the first column of the next line. If this would take the cursor below the bottom margin, the screen rolls up one line and the cursor is positioned on the first character of the bottom line.

CURSOR BACKWARD

ESC[PnD Default value: 1

This sequence moves the active position to the left. The distance moved is determined by the parameter. A parameter value of 0 or 1 moves the active position one position to the left. A parameter value of n moves the active position n positions to the left. If an attempt is made to move the cursor to the left of the left margin, the cursor moves to the last column in the previous row. If this would place the cursor above the home position, the cursor does not move.

CURSOR POSITION

ESC[P1;PcH or Default value: 1
ESC[P1;Pcf

This sequence moves the cursor position to the position specified by the parameters.

P1=Line number. A parameter value of 0 or 1 moves the active cursor position to the first line in the display. A parameter value of n moves the active position to the nth line in the display. If n>25, the system treats n as 25.

Pc=Column number. A parameter value of 0 or 1 moves the active cursor position to the first column in the display. A parameter value of n moves the active position to the nth column. If n>80, the system treats n as 80.

SELECT CHARACTER ATTRIBUTES

ESC[Ps;...;Psm

This escape sequence sets character attributes. Once the sequence is executed, all characters transmitted afterwards are rendered according to its parameters until the escape sequence is used again.

Parameter	Meaning	
	Attributes off (default: green color, color monitor)	
1	Attributes off (default: green color)	
2	Vertical line	
3	Over line	
4	Under line	
5	Blink	
6	Not used	
7	Reverse	
8-15	Not used	
16 30	Secret	
17 31	Red color/Highlight*	
18 34	Blue color	
19 35	Purple color	
20 32	Green color (default)	Color Parameters
21 33	Yellow color	
22 36	Light blue color	
23 37	White color	

*Only the Highlight attribute is available for the monochrome CRT.

NOTE

The color and secret parameters are mutually exclusive. If neither color nor secret is specified, the green color default is used.

The attributes off parameter (Ps=0 or 1) cannot be specified with other parameters. If it is, it is ignored.

ERASE WITHIN DISPLAY

ESC[PsJ Default value: 0

This sequence erases some or all of the characters in the display according to the specified parameter.

Parameter	Meaning
0	Erase from the active position to the ènd of the screen.
1	Erase from the start of the screen to the active position.
2	Erase all of the display. All lines are erased, and the cursor does not move.

ERASE WITHIN LINE

ESC[PsK Default value: 0

Erases some or all characters in the active line according to the specified parameter.

Parameter	Meaning
0	Erase from the active position to the end of the line.
1	Erase from the start of the screen to the active position.
2	Erase all of the line.

AUXILIARY CHARACTER SET

ESC(1

This function is used to access the auxiliary character codes (20H - FDH) created by the Auxiliary Character Generator program (CHR external command). The one character immediately following the command is treated as the auxiliary character code. In Direct Console I/O (Function Request 06H) the available auxiliary character codes have a range of 00H to FFH.

NOTE

The character immediately following ESC is the open parentheses character, (, not the square bracket.

For more information on the Auxiliary Character Generator program, refer to the *MS-DOS System Programmer's Guide.*

SET A MODE

ESC[Psh

This sets the mode specified by the parameter. Only the values listed below may be used; all others are ignored.

Parameter	Meaning
1	Disable system status display
2	Disable key click
5	Disable cursor display
7	Disable keyboard input

RESET A MODE

ESC[Psl

This escape sequence resets the mode specified by the parameter. Only the values listed below may be used; all others are ignored. The final character is the lowercase letter l, not the number one.

Parameter	Meaning
1	Enable system status display
2	Enable key click
5	Enable cursor display
7	Enable keyboard input

DEVICE STATUS REPORT

ESC [6 n

The console driver will output a Cursor Position Report (CPR) sequence on receipt of a Device Status Report sequence (DSR).

CURSOR POSITION REPORT

ESC [P ; Po R

The Cursor Position Report (CPR) sequence reports the current cursor position via standard input (console driver). The first parameter specifies the current line and the second parameter specifies the current column.

SAVE CURSOR POSITION

ESC [s

The Save Cursor Position (SCP) sequence saves current cursor position. This cursor position can be restored with the Restore Cursor Position (RCP) sequence.

RESTORE CURSOR POSITION

ESC [u

The Restore Cursor Position (RCP) restores the cursor position to the value it had when the console driver received the SCP sequence.

Note that the Device Status Report escape sequence performs the same task as the Report Cursor Position escape sequence.

ADM-3A MODE CURSOR POSITION ESCAPE SEQUENCE

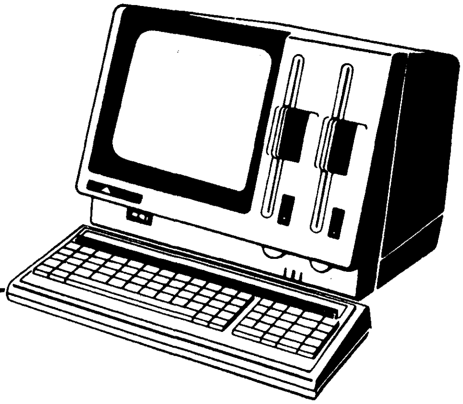
ESC = lc

This escape sequence function is compatible with that used by the Lear Siegler ADM-3A terminal.

This sequence moves the cursor position to the position specified by the parameters.

l = Line number. The line number is a binary value in the range 20H (first line) -38H (25th line). If l=38H, the system treats l as 38H. If l >20H, the system treats l as 20H.

c = Column number. The column number is a binary value in the range. 20H (first column)-6FH (80th column). If c=6FH, the system treats c as 6FH. If c >20H. the system treats c as 20H.



Chapter 5

MS-DOS Graphics Supplement

The MS-DOS Graphics Supplement provides a powerful interface between the APC graphics hardware and applications running under MS-DOS. The supplement consists of a Pascal unit called Graf_Draw. Procedures perform tasks such as drawing lines, circles, rectangles and arcs, displaying graphics texts, polygon filling, pattern generation, and character font generation.

To use the supplement, the following minimal APC system configuration must be available:

- one or more diskette drive(s)
- 256K bytes or more of RAM
- an APC monochrome or color graphics board.

The following graphics application files are supplied on the MS-DOS system diskette:

FNTCOMP.EXE — The Character Font Compiler, which allows user-designed character fonts to be created and stored for later use by applications programs.

PATCOMP.EXE — The Area Fill Pattern Compiler, which allows user-defined patterns to be used in filling polygon areas. The patterns can be stored for later use by applications programs.

PRC0.OBJ, GRIMPL.OBJ — Two object modules containing the graphics procedures used by application programs. These modules are combined by LINK.EXE with user applications to produce an executable program.

GRINTE.PAS File containing the Pascal constant, type, variable, and external procedure declarations for the Graf_Draw unit. It must be copied, using the Include compiler directive, into a Pascal source file that uses the Graf_Draw unit. (See GPTEST.PAS for an example of this Include.)

GPTEST.PAS The validation suite for the supplement. This file contains the Pascal source code, which can be used as an example of the way the procedures of the Graf_Draw unit work.

KEYBRD.ASM Assembly language module used by the GPTEST program to gain direct access to the APC keyboard.

GPTEST.EXE The executable file for the validation suite. It can be executed as a demonstration of the Graf_Draw unit. GPTEST.EXE can also be used to verify proper functioning of the graphics hardware.

FONT01.TXT The source file for the standard character font. This file is also an example of the input format for the font compiler. It defines characters of 16 pixels by 16 pixels.

FONT01.FNT The file, written by the Font Compiler, that contains the "object" code for the standard character font. This file is used at run time when the application requests that character data be displayed.

PAT00.PAT A source file processed by the Pattern Compiler to produce a pattern data file that can be used to fill areas on the graphics display. The file is also an example of the pattern source file format. It defines a 16 x 16 pixel blue grid pattern.

PAT01.TXT A source file for a 16 x 16 pixel blue and green grid.

PAT01.PTN The data file for the above pattern.

PAT02.TXT A source file for an 8 x 8 pixel red triangle pattern.

PAT02.PTN The data file for the above pattern.

PAT03.TXT A source file for a 11 x 11 pixel green triangle pattern.

PAT03.PTN The data file for the above pattern.

PAT04.TXT A source file for a 10 x 10 pixel blue triangle pattern.

PAT04.PTN The data file for the above pattern.

EXECUTING THE GRAPHICS TEST

GPTEST.PAS demonstrates the capabilities of the Graphics Supplement and verifies that the graphics hardware of the APC is working properly.

To begin, insert the MS-DOS system diskette, containing the file GPTEST.PAS, the font data file FONT01.FNT, and the pattern data files (PAT00.PTN, PAT01.PTN, and so on), into drive A. Enter GPTEST to start the test.

When the test program begins, it will prompt for three entries:

- The first entry is for the background color. Enter the index of the color you want to be used as a background throughout the test. Use black (color) for best results with this test program. Note that you will enter the number of the color, not the name, for this prompt.
- The second entry is for the pattern to use for the area fill operations. Enter a number, 0 to 4, to select the .PTN data file containing the fill pattern you want. Entering 0 selects PAT00.PTN, 1 selects PAT01.PTN, and so on. Note that patterns containing colors other than green will not display on a monochrome graphics APC. Therefore, for a monochrome system, it is suggested that pattern number 1 or 3 be selected.
- The final entry is the number of the font (.FNT) file containing the character display font to be used for the text portion. Enter 1 to select the .FNT file FONT01.FNT.

At this point the graphics test begins execution. There are several subtests within the GPTEST program . Before each one, you will be asked whether or not you want to run the subtest. For example, before the first subtest, you will be asked "Test Cursor (Y/N/esc) ?" Type Y to execute the cursor subtest or N to skip to the next subtest. Press ESC to exit GPTEST and return to the system prompt.

Several times within each subtest, a display will appear for some function or combination of functions, and will remain on the screen until you press RETURN. This gives you time to inspect the results of each function. At these times the prompt "type return to continue" will appear in the lower left corner of the screen.

This prompt may be difficult to see during some displays and with certain nonblack background colors. If the display seems to be inactive for more than about ten seconds, chances are that the "type return to continue" prompt is displayed but invisible because of the colors displayed on top of it. Pressing RETURN will allow the test to continue.

Many of the displays of the GPTEST program contain colors that will not display on monochrome APCs. For this reason, many of the functions will appear to do nothing. Only displays (and portions of displays) that use green will be seen on a monochrome APC.

USING THE GRAF_DRAW UNIT

To gain access to the Graph_Draw unit, a Pascal application program must do the following:

- Use the $Include compiler directive to copy the GRINTE.PAS file into the application. This provides the interface declarations of the Graf_Draw unit: $INCLUDE 'GRINTE.PAS'
- Include the Uses statement to gain access to the Graf_Draw procedures: USES GRAF_DRAW;

  The application source program may make use, only once, of the Core Record and Graf_Draw procedures described in the following section.

  Before execution, you must link the actual object code for the Graf_Draw procedures and data areas with the object code resulting from the compilation of the application program. To do this, execute LINK.EXE (the MS-LINK Linker Utility™) with GRIMPL.OBJ and PRC0.OBJ, which are supplied as modules to be linked to the module (or modules) containing the application program. For example, when linking the GPTEST.EXE, the following LINK input file could be used.

```
gptest prc0 grimpl keybrd/m/l
gptest
gptest
;
```

Figure 5-1 is a flow diagram of the graphics application development process.

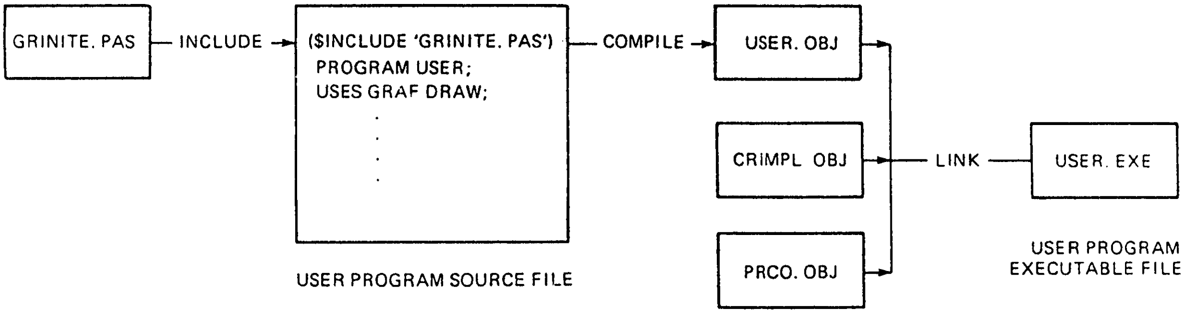


Figure 5-1 Graphics Application Development Process

THE GRAF_DRAW UNIT

The Graf_Draw unit is composed of 28 procedures written in Pascal. This program works through a segment of code called GRINTE.PAS that acts as an interface between Graf_Draw and applications programs. This interface segment defines values for the next operation and a record, called the Core Record. The Core Record contains variables describing the current state of the graphics system.

THE INTERFACE UNIT

The Interface unit, GRINTE.PAS, contains the following code. Among the values designated in this program segment are constants, for example, the display screen size and variables, such as the font type used by the application program. Variables are defined in the Core Record (see the Core_Record variable).

```
Const Graf_Version = '0.4;

Type Cur_Attribute  = (Cur Disable,       Cursor disable
                       Cur_Enable,        Cursor enable
                       Cur_Visible,       Cursor visible
                       Cur_Invisible,     Cursor invisible
```

```
                    Cur_Small,              Cursor small
                    Cur_Full;               Cursor full
Switch_Types    = (Off, On);
Overlay_Type    = (Xor Mode,                Replace contents
                    Replace);               Merge contents
Display_Type    = (Fast,                    No filling
                    Fill);                  Fill all polygons
Int_Type        = (Plain,                   Solid rectangle
                    Patterned);             User pattern
Edge Type       = (Solid Line,              Solid border
                    Interior);              Invisible border

Directions      = (Left,                    Left direction or position
                    Right,                  Right direction or position
                    Up,                     Up direction
                    Down,                   Down direction
                    Top,                    Top position
                    Center,                 Center (horz/vert) position
                    Bottom;                 Bottom position
Font_Type       = -1..99;
Pat_Type        = -1..99;
Color_Index     = 0..15;
Point           = Integer
Point_Array     = Array 1..128 Of Point;
Sorcery         = Integer;

Core_Record     = Record
                      X_Min,                    Left edge of screen
                      X_Max,                    Right edge of screen
                      Y_Min,                    Top edge of screen
                      Y_Max,                    Bottom edge of screen

                      X_Org,                    X-origin of fill pattern
                      Y_Org,                    Y-origin of fill pattern

                      X_CP                      X-current position
                      Y_CP       :Point;        Y-current position
                      Line_Index,               Line color
                      Fill_Index,               Filled object color
                    Text_Index,       Text color
                    Background        :Color_Index      Background color

                    Line_Style        :Integer;         Line Pattern
                    Display_Mode      :Display_Type     Fast/Fill
                    Overlay_Mode      :Overlay_Type     Replace/Xor pixel
                    Polygon_Interior  :Int_Type         Plain/Patterned
                    Polygon_Edge      :Edge_Type        Solid_Line/Interior

                    Font_Number       :Font_Type;       Current font numbers
                    Font_Cols                           Columns per char
                    Font_Rows         :Point,           Rows per char

                    Char_Spacing      :Real             Character pitch
                    Top_Bottom                          Above/below text
                    Left_Right,                         Left/right of text
                    Char_Path         :Directions;      Write direction
                    DX_Charup,                          Char rotation X
                    DY_Charup         :Integer;         Char rotation Y
                    Char_Height,                        Rows to display
                    Char_Width        :Integer          Columns to display

                    Pat_Number        :Pat_Type         User pattern number
                    Pat_Cols,                           Columns in pattern
                    Pat Rows          :Point;           Rows in pattern

                    File_Prefix       :String 1         Prefix for font text

Var Core : Core_Record;

Procedure Move_Abs (X_Position,
                    Y_Position : Point);

Procedure Move_Cursor (X_Position,
                       Y_Position : Point)

Procedure Move_Rel (Delta_X,
                    Delta_Y : Point);

Procedure Set_Cursor (Attrib : Cur_Attribute);

Procedure Size_Cursor (Size : Integer);

Procedure Set_Fill_Pattern (Pattern_Num : Pat_Type);

Procedure Box_Abs (X_Corner, Y_Corner : Point);
Procedure Box_Rel (Width, Height : Point);
Procedure Write_Block_Pixels (Data : Sorcery; Rows, Columns : Integer);
Procedure Read_Block_Pixels (Data : Sorcery; Rows, Columns : Integer);
Procedure Set_Charup (DX_Charup,
                      DY Charup : Integer);
Procedure Set_Font (Font_Num : Font Type);
Procedure Text (The_String : String);
Procedure Set_Line_Style (Dot_1,
                          Dot_2,
                          Dot_3,
                          Dot_4,
                          Dot_5,
                          Dot_6,
                          Dot_7,
                          Dot_8 : Switch_Type);
Procedure Line_Abs (X_End,
                    Y_End : Point);
Procedure Line_Rel (X_Length,
                    Y_Length : Point);
Procedure PLine Abs (Var X_End,
                         Y_End : Point Array;
                         Count : Integer);
Procedure PLine Rel (Var X_Length,
                         Y Length : Point Array;
                         Count : Integer);
Procedure Circle Abs (X_of_Edge, Y_of_Edge : Point);
Procedure Circle_Rel (Radius : Point);
Procedure Define_Color (Index,
                        Red,
                        Green,
                        Blue,
                        Blink,
                        Hard_Copy : Integer);
Function Inq_Value (Option : Integer) : Integer;
Procedure Plane_Enable (Planes : Integer);
Procedure Plane_Visible (Planes : Integer);
Procedure Set_Palette (Pal_Name : String);
Procedure Set_Value (Opcode,
                     Value : Integer);
Procedure Erase;
Procedure Erase_Alpha;
Procedure Flood;
Procedure Arc_Rel (Radius : Integer;
                   Start_Angle,
                   End_Angle : Real;
                   X_Start,
                   Y_Start,
                   X_End,
                   Y_End : Integer
Procedure Arc_Abs (Var Radius : Integer;
               Var Start_Angle,
                   End_Angle : Real:
                   X_Start,
                   Y_Start,
                   X_End,
                   Y_End :       Integer;
```

Table 5-1 lists the names, initial values, and a brief description of the Core Record fields.

Table 5-1 Core Record Fields

VARIABLE	INITIAL VALUE	DESCRIPTION
X_ Min	0	* Left edge of the screen
X_Max	639	* Right edge of the screen
Y_Min	0	* Top of the screen
Y_Max	479	* Bottom of the screen
X_Org.		** X-origin of fill pattern
Y_Org.		** Y-origin of fill pattern
X_CP	0	X current position
Y_CP	0	Y current position
Line_Index	7 (white)	Color of line
Fill_Index	7 (white)	Color of filled object
Text_Index	7 (white)	Color of text
Background	0 (black)	Color of background
Line_Style	?	Line pattern
Display_Mode	Fast (no fill)	Switch for filling: Fast/Fill
Overlay_Mode	Replace	Switch for Replace/Xor Pixels
Polygon_Interior	Plain	Fill type: Plain/Patterned
Polygon_Edge	?	Filled object border Solid Line/Interior
Font_Number	-1 (undefined)	** Current font number
Font_Cols	?	** Width of font
Font_Rows	?	** Height of font
Char_Spacing	?	Spacing between characters
Char_Path	?	Direction of character string
DX_Charup	?	Character rotation in the X direction
DY_Charup	?	Character rotation in the Y direction
Char_Height	?	Height of text characters
Char_Width	?	Width of characters
Pat_Number	-1 (undefined)	** Current fill pattern
Pat_Cols	?	** Width of current pattern
Pat_Rows	?	** Height of current pattern
File_Prefix	?	Volume where .FNT and .PTN files are located

NOTES:

The single asterisk (*) denotes variables set once by the system.

The double asterisks (**) indicate variables that are automatically set by procedure calls.

You should not attempt to set the values of the Core Record (CORE.) variables flagged by * and ** in programs. If you do, the results are unpredictable.

TERMS THAT DESCRIBE SCREEN DISPLAYS

The following terms describe elements of the APC graphics display. These terms are used in explanations of the Graf_Draw procedures, FNTCOMP.EXE, and PATCOMP.EXE.

Color The APC can display eight colors. These colors are fixed and cannot be altered. The term "index" is used as a synonym for "color" in many places in this text. A color index is a pointer into a color table that determines which color is to be used for drawing lines, shapes, displaying text, and so on. In reality, since the color scheme is fixed, there is no need to keep such tables around. Therefore, the color tables are conceptual only, and the index is the color.

The color indexes are as follows:

0 - Black
1 - Red
2 - Green
3 - Yellow
4 - Blue
5 - Magenta
6 - Turquoise
7 - White

CP The system's current position (CP). The point within the graphic coordinate space where the next output operation will take place. The CP is kept in memory in the X CP and Y_CP CORE. variables.

In this manual, the CP is occasionally indicated by an ordered pair of X and Y coordinates, such as (100, 200).

Some of the Graf_Draw procedures have an effect on the value ot tne CP, others do not. This effect is indicated in this discussion by the following expression:

```
CP -- (New_X_Value, New_Y_Value);
```

Cursor A software controlled graphics cursor. It shows on the screen as a hairline cross with equal vertical and horizontal bar sizes. The cursor can be any size, up to that of the full screen. The default cursor size is 15 pixels.

Pixel The elementary display unit. Each pixel is a dot (approximately 1/10 inch) on the APC screen. It is individually controlled by attributes stored for it in the graphics display memory. The APC has a 640 x 480 pixel display. The attributes for each pixel are stored in a four-bit field, where the high order bit is always 0 and the remaining three bits give the color associated with the pixel.

Plane The display screen may be visualized as three superimposed bit planes, one for each of the primary colors: red, green and blue. The color for an individual pixel is therefore determined by a three-bit value, depicting a color value or index.

GRAF_DRAW UNIT PROCEDURES

The Graf_Draw unit procedures are described in the following pages. For each procedure, you are given

- the complete procedure declaration with its parameters
- a description of what the procedure does
- a sample call with an explanation of the associated effect.

PROCEDURE MOVE_ABS

Declaration:

Procedure Move_Abs(X_Position, Y_Position : Point);

Description:

This procedure sets the CP to the new position given by the values in X_Position and Y_Position.

Effect on CP:

CP -- (X_Position, Y_Position);

Example:

MOVE_ABS (50,100);

This example sets CORE.X_CP to 50 and CORE.Y_CP to 100.

PROCEDURE MOVE_REL

Declaration:

Procedure Move_Rel(Delta_X, Delta_Y : Point);

Description:

This procedure changes the value of the current position of the variables. The parameters Delta_X and Delta_Y are added algebraically to the values of CORE.X_CP and CORE.Y_CP respectively.

Effect on CP:

CP — (X_CP+Delta_Y);

Example:

MOVE_REL (10,20);

This example moves the current position of X to CORE.X_CP + 10 and of Y to CORE.Y_CP + 20.

PROCEDURE SET_CURSOR

Declaration:

Procedure Set_Cursor (Attrib : Cur_Attribute);

Description:

The various attributes for the graphics cursor are set via this procedure. Cursor attributes and their effects are as follows.

Cur_Disable — The cursor is disabled. All further cursor commands will be ignored.

Cur_Enable — The cursor is enabled. Subsequent cursor commands will be honored.

Cur_Visible — If the cursor is enabled, it will be made visible.

Cur_Invisible — If the cursor is enabled, it will be made invisible. While invisible, all other cursor commands can still be used but the effects will not be apparent until the cursor is made visible again.

Cur_Small — The cursor is set to a default size of 15 pixels.

Cur_Full — The cursor is set to the size of the screen.

Effect on CP:

None.

Example:

SET_CUR (Cur_Invisible);

This example turns the cursor invisible so that it may be moved around the screen or have its size changed before it is made visible again.

PROCEDURE SIZE_CURSOR

Declaration:

Procedure Size_Cursor (Size : Integer);

Description:

This procedure sets the size of the graphics cursor. The size is given in pixels and can be changed only if the cursor is enabled.

Effect on CP:

None.

Example:

SIZE_CURSOR (30);

This example results in the graphics cursor being drawn with lines that are 30 pixels long.

PROCEDURE SET_FILL

Declaration:

Procedure Set_Fill_Pattern (Pattern_Num : Pat Type);

Description:

When drawing boxes, circles, and other shapes, you may use user-defined patterns to fill these areas. This procedure is used to select one of the defined patterns.

The value of the parameter must correspond to a disk file generated by the Pattern Compiler (see the section THE PATTERN COMPILER for details). The file containing the pattern must be named PAT*.PTN where * is a number between 0 an 99.

The variables CORE.Pat_Number, CORE.Pat_Rows and CORE.Pat_Cols are set by this procedure.

Effect on CP:

None.

Example:

SET_FILL_PATTERN (3);

This example causes the system to read the file PAT03.PTN if it is present. All future pattern fills will use this pattern.

PROCEDURE BOX_ABS

Declaration:

Procedure Box_Abs (X_Corner : Point);

Description:

This procedure draws a rectangular box starting at the CP. The box is drawn parallel to the X and Y axes. One corner is located at the CP, and the opposite corner at the point given by X_and Y_Corner.

If CORE.Display_Mode = Fast, the box will be drawn as a rectangular outline. If it is Fill, the box will be drawn as a rectangular solid.

If CORE.Polygon_Edge = Solid Line and CORE.Display_Mode = Fill, the box will be drawn as a solid rectangle with a border. If it is Interior, no border will be drawn.

If CORE.Polygon_Interior = Plain and CORE.Display_Mode = Fill, the box will be drawn as a solid-colored rectangle. If CORE. Polygon_Interior is Patterned, then the box will be drawn using the current pattern.

If CORE.Overlay_Mode = Replace, each pixel on the screen will be overwritten by the corresponding pixel of the box. If it is XOR, then a Boolean XOR is performed for the screen and the box and the result is displayed.

CORE.Line_Index specifies the color in which the border of the box is drawn.

CORE.Fill_Index specifies the color to be used for a solid fill.

Effect on CP:

The CP retains the value it had before the box was drawn.

Example:

```
CORE.Line_Index := 1;
CORE.Fill_Index := 4;
CORE.Displaymode := Fill;
CORE.Polygon_Interior := Plain;
CORE.Polygon_Edge_ := Solid Line
CORE.Overlay_Mode := Replace;

Box_Abs(90,70);
```

This example draws a box with a border color of 1 and fills it with a solid color of 4. The box starts at the CP and has its opposite corner at (90,70).

PROCEDURE BOX_REL

Declaration:

Procedure Box Rel (Width, Height : Point);

Description:

This procedure is similar to BOX_ABS. The only difference is that the point defining the corner of the box opposite to the anchor point is given as a relative displacement from the CP. Therefore, width is an offset from the current X position and height is an offset from the current Y position.

Effect on CP:

See BOX_ABS.

Example:

Refer to BOX_ABS.

PROCEDURE WRITE_BLOCK_PIXELS

Declaration:

Procedure_Write_Block_Pixels (Data : Sorcery; Rows, Columns : Integer);

Description:

This procedure writes a rectangular array of pixels from a user-defined area to the screen starting at the CP. The parameters Rows and Columns define the size of the pixel array to be transferred from memory. The order of display is from left to right and bottom to top.

The memory array resides in an area pointed to by the Data parameter. This parameter is of the Sorcery type and needs to be set prior to the procedure call.

CORE . Overlay_Mode has an effect on this function if it is set to XOR.

Effect on CP:

None.

Example:

```
Var Screen_Seg : Packed array [0..3000) of boolean;
        Data : Integer;
begin
  move left(Screen_Seg, Data,2); (*Move the address of the
  screen into Data*)

  Move Abs (100,100);
  Write block Pixel (Data, 20,20);
end;
```

This example will write the pixels from Screen_Seg to the screen starting at (100, 100). The rectangular screen area that is affected by this code is 20 pixels on each side.

PROCEDURE READ_BLOCK_PIXELS

Declaration:

Procedure Read_Block_Pixels (Date : Sorcery; Rows, Columns : Integer);

Description:

This procedure does just the opposite of WRITE_BLOCK_PIXELS. It writes a rectangular array of pixels starting at the CP from the screen to a user-defined area. The parameters mean the same thing, except that Data is now the destination for the screen area defined by the current position and the parameters, Rows and Columns.

Effect on CP:

None.

Example:

Refer to WRITE_BLOCK_PIXELS.

PROCEDURE SET_CHARUP

Declaration:

Procedure Set_Charup (DX_Charup, DY_Charup : Integer);

Description:

This procedure establishes the rotation angle for each character output via subsequent TEXT calls. It does not specify the direction for the character path (given by the contents of CORE.Char_Path).

The rotation angle is determined by a normalized Cartesian vector system and is governed by the following variables:

DX_Charup	DY_Charup	Character Rotation
0	1	Right side up
0	-1	Upside down
1	0	Rotated to the right
-1	0	Rotated to the left

If DX_Charup and DY_Charup have values other than (-1,0,1), the system automatically normalizes the vector based on the larger of the two values.

Effect on CP:

None.

Example:

SET_CHARUP(-1,0);

This example causes all characters output by subsequent TEXT calls to appear rotated to the left.

PROCEDURE SET_FONT

Declaration:

Procedure Set_Font (Font_Num : Font_Type);

Description:

This procedure selects a user-defined text font for use in the TEXT procedure. The file containing the pattern must be named FONT*.FNT, where "*" is a number between 0 and 99. The variables CORE . Font_Number, CORE . Font_Rows, and CORE . Font_Cols are set by this procedure.

Effect on CP:

None.

Example:

SET_FONT (3);

This example causes the system to read the file FONT03. FNT if it is present. All future calls to the TEXT procedure will use this font.

PROCEDURE TEXT

Declaration:

Procedure Text (The_String : string);

Description:

This procedure writes a string of text to the screen using a user-defined font. The size of the characters, their orientation the spacing between them, and their paths can be defined. The parameter is a standard Pascal string to be displayed.

CORE . Char_Width and CORE . Char_Height define the size of the characters to be printed, rounded to the nearest multiples.

CORE . Char_Path defines the direction in which the text string is to be written (Left, Right, Up or Down).

CORE . DX_Charup and CORE . DY_Charup define the rotation at which the characters are written. These variables can be set with the SET_CHARUP procedure.

CORE .Char_Spacing defines the distance between characters. This is a real number and is used to represent a unit of the character size. The number can be a fraction (for example, .5 to move characters one-half a character space apart), or it can be a negative number to move the characters closer together.

CORE .Left_Right and CORE . Top_Bottom are used to position the text relative to the current position (X_CP,Y_CP). CORE .Left_Right is used to position the string so that the "left" edge, "right" edge or "center" of the string is located on the X component of the current position. CORE . Top_Bottom is used to position the string so the top edge, bottom edge, or center is located on the Y component of the current position.

CORE .Text_Index specifies the color for the string.

CORE Font_Number is set by the procedure SET_FONT and is the number of the current text font.

Effect on CP:

None.

Example:

```
CORE .Text_Index : 1;
CORE .Char_Width : 12;
CORE .Char_Height : 30;
SET_FONT (1)
TEXT ('LETS SEE WHAT THIS LOOKS LIKE')
```

PROCEDURE SET_LINE_STYLE

Declaration:

Procedure Set_Line_Style (Dot_1, Dot_2, Dot_3, Dot_4, Dot_5, Dot_6, Dot_7, Dot_8 : Switch_Type);

Description:

By this procedure, you define the type of line that will be used to draw lines, circles and boxes. It can be a solid line, a dashed line, or a line with dots and dashes. You define one segment of the line which is composed of eight pixels. Each of the eight pixels can be turned either "ON" or "OFF."

Effect on CP:

None.

Example:

SET_LINE_STYLE(ON,ON,ON,OFF,OFF,OFF);

This example creates a line that will have 4 pixels on, then 4 pixels off, then 4 on, then 4 off, and so on.

PROCEDURE LINE_ABS

Declaration:

Procedure Line_Abs (X_End, Y_End : Point);

Description:

This procedure draws a line from the CP to the point defined by X_End and Y_End. The current position is then updated to the X_End, Y_End position.

If CORE . Overlay_Mode = Replace, each pixel on the screen will be overwritten by the corresponding pixel of the line. If this variable equals XOR then a Boolean XOR of the screen and the line will be performed and the result will be displayed.

CORE . Line_Index is the color in which the line will be drawn.

CORE . Line_Style is set by the SET_LINE_STYLE procedure.

Effect on CP:

CP — (X_END, Y_End);

Example:

```
SET_LINE STYLE(ON,ON,ON,ON,OFF,OFF,OFF,OFF);
CORE .Line_Index := 1;
CORE . Overlay_Mode := xor;
MOVE_ABS(100,100);
LINE_ABS(120,120);
```

PROCEDURE LINE_REL

Declaration:

Procedure Line_Rel (X_Length, Y_Length : Point);

Description:

This procedure is the same as LINE_ABS except that the end point is specified by relative displacements from the CP.

Effect on CP:

CP — (X_CP+X_Length, Y_CP+Y_Length);

Example:

```
SET_LINE_STYLE(ON,ON,ON,ON,OFF,OFF,OFF,OFF);
CORE . Line_Index : 1;
CORE . Overlay_Mode : XORE;
MOVE_ABS(100,100);
LINE_REL (21,21);
```

This example performs the same operation as the one in the example for LINE_ABS.

PROCEDURE PLINE_ABS

Declaration:

Procedure Pline_Abs (Var X_End, Y_End ; Point_Array; Count ; Integer);

Description:

This procedure draws a series of lines from the CP to the first set of points in the two arrays X_End and Y_End. Then it draws the next line to the second position in the array and so on for "Count" lines. At the end, the CP is pointing to the end of the last line. A line of zero length implies a pen-up command, so the next line is interpreted as cursor movement only with no display. The line after that will be displayed again.

If CORE .Overlay_Mode = Replace, each pixel on the screen will be overwritten by the corresponding pixel of the line. If it is XOR, then a Boolean XOR of the screen and the line will be performed and the result will be displayed.

CORE .Line_Index is the color in which the line will be drawn.
CORE .Line_Style is set by the procedure SET_LINE_STYLE.

Effect on CP:

CP —(X_End[Count], Y_End[Count]);

Example:

```
CORE .Line_Index := 1;
X_END[1]  := 200;
Y-END[1]  := 100;
X_END[2] := 200;
Y_END[2] := 200;
X_END[3] := 100;
Y_END[3] := 200;
X_END[4] := 100;
Y.END[4]  := 100;

MOVE ABS (100,100);
PLINE_ABS(X_END, Y_END, 4);
```

The above example will draw a box in color 1 starting at (100,100) and returning there.

PROCEDURE PLINE_REL

Declaration:

Procedure Pline_Rel (Var X_Length, Y_Length : Point_Array; count : Integer);

Description:

This procedure is the same as PLINR_ABS except that the lines are specified in terms of relative displacements rather than absolute end point locations. A line of length zero still implies a pen-up command.

Effect on CP:

CP — (X_Final, Y_Final);

```
     Where:
        X_Final = Y_CP+X Length [l]+X_Length[2]+
        ... +X_Length[COUNT]
     Y_Final = Y_CP+Y_Length[2]+ ... +Y_Length [Count]
```

Example:

```
CORE .Line Index := l;
X_LENGTH[1] := 100;
Y_LENGTH[1] := 0;
X_LENGTH[2] := 0;
Y_LENGTH[2] := 100;
X_LENGTH[3] := -100
Y_LENGTH[3] := 0;
X_LENGTH[4] := 0;
Y_LENGTH[4] := -100;
MOVE ABS (100, 100);
PLINE_REL(X_LENGTH, Y_LENGTH, 4);
```

This example will draw a box in color l starting at (100,100) and returning there.

PROCEDURE CIRCLE_ABS

Declaration:

Procedure Circle_Abs (X_of_Edge Y_Of_Edge : Point);

Description:

This procedure draws a circle centered around the CP with its edge passing through the point defined by X_Of_Edge and Y_Of_Edge. The circle can be drawn as an outline or as a solid disk. If it is solid, it may be filled with a solid color or a user-defined pattern.

If CORE .Display_Mode = Fast, the circle will be drawn as an outline. If the variable = Fill, the circle will be drawn as a solid.

If CORE . Polygon_Edge = Solid Line and CORE .Display_Mode = Fill, the circle will be drawn as a solid disk with a border. If it is Interior, no border will be drawn.

If CORE .Polygon_Interior = Plain and CORE .Display_Mode = Fill, the circle will be drawn as a solid-colored disk. Polygon_Interior is Patterned, then the circle will be drawn using the current pattern.

If CORE .Overlay_Mode = Replace, each pixel on the screen will be overwritten by the corresponding pixel of the circle. If the variable is XOR, then a Boolean XOR is performed with the current contents of the screen before the result is displayed.

CORE .Line_Index specifies the color in which the border of the circle is drawn.

CORE .Fill_Index specifies the color to be used for a solid fill.

Effect on CP:

The CP retains the value it had before the procedure call.

Example:

```
CORE .Line_Index := l;
CORE .Fill_Index :=4;
CORE .Display_Mode := Fill;
CORE .Polygon_Edge := Solid Line;
CORE .Polygon_Interior := Plain;
CORE .Overlay_Mode := Replace;

CIRCLE_ABS (90,70);
```

This example draws a circle with a border color of l and fills it with a solid color of 4. The circle's center is at the current position and its border passes through the point (90, 70).

PROCEDURE CIRCLE_REL

Declaration:

Procedure Circle_Rel (Radius : Point);

Description:

This procedure is similar to the CIRCLE_ABS procedure. The difference is that CIRCLE_REL draws a circle centered at the CP whose border is drawn "Radius" pixels away from the center, not through a specific point.

Effect on CP:

The CP retains the value it had before the procedure call.

Example:

Refer to CIRCLE_ABS.

PROCEDURE DEFINE_COLOR

Declaration:

Procedure Define_Color (Index, Red, Green, Blue, Blink, Hard_Copy : Integer);

This procedure is not implemented.

PROCEDURE INQ_COLOR

Declaration:

Procedure Inq_Color(Var Index, Red, Green, Blue, Blink, Hard_Copy:Integer);

Description:

Given the index of a color in Index, DEFINE_COLOR will set the remaining variables according to that color's internal composition.

Red, Green, and Blue indicate the amount of each of the primary colors that is used to make up a particular color among the eight available. The value 0 indicates the absence of a primary color, and the value 3 indicates 100% usage of a primary color. These are the only values that are currently used by the APC hardware.

The Blink and Hard_Copy options are not implemented in this version of the Graphics Supplement.

This procedure will always return the same values for a given color.

Color Index	Color	R G B	Blink	Hard_Copy
0	Black	0 0 0	0	0
1	Red	3 0 0	0	0
2	Green	0 3 0	0	0
3	Yellow	3 3 0	0	0
4	Blue	0 0 0	0	0
5	Magenta	3 0 3	0	0
6	Turquoise	0 3 3	0	0
7	White	3 3 3	0	0

Effect on CP:

None.

Example:

Index := 2;
INQ_COLOR(Index, Red, Green, Blue, Blink Hard.Copy)

The above example will return the following values:

Color	Value
Red	0
Green	3
Blue	0
Blink	0
Hard_Copy	0

PROCEDURE INQ_VALUE

Declaration:

Procedure Inq_Value (Option : Integer) : Integer;

Description:

This procedure returns information on the type of monitor that is being used. The Option parameter should always be set to 0. Currently, the result of this function is always 0, indicating an APC with a 640 x 480 pixel monitor. This function will be enhanced in future releases.

Effect on CP:

None.

Example:

Machine := INQ_VALUE(0);

This example will set the Machine variable to 0.

PROCEDURE PLANE_ENABLE

Declaration:

Procedure Plane_Enable (Planes : Integer);

Description:

This procedure sets a binary mask that controls values written to the system's graphics memory. The mask is set to the binary equivalent of the value in the Planes parameter.

Initially the mask is -l (Hex FFF), which lets every color value go out unchanged. Different values of the mask will ultimately result in the suppression on one or more of the primary colors from the final pixel color. Before a value is written to graphics memory, it is first ANDed with the current value of the graphics output mask.

Effect on CP:

None

Example:

PLANE_ENABLE(6);

This example will mask out the low-order bit of every pixel value written to memory. Therefore, the Blue component will be suppressed.

For additional details, consult the DEFINE_COLOR procedure description and the section THE PATTERN COMPILER.

PROCEDURE PLANE_VISIBLE

Declaration:

Procedure Plane_Visible (Planes : Integer);

This procedure is not implemented.

PROCEDURE SET_PALETTE

Declaration:

Procedure Set_Palette (Pal_Name : String);

This procedure is not implemented.

PROCEDURE SET_VALUE

Declaration:

Procedure Set_Value (Opcode, Value : Integer);

Description:

This procedure performs internal control functions and directly sets some of the Core Record variables. It may prove useful in situations where bypassing the procedure calling overhead is critical to system performance. It is recommended, however, that you use the standard procedures that accomplish the desired result wherever possible.

The following is a list of acceptable commands.

Opcode	Value	Operation
0	0	Initializes the graphics hardware and various flags. This is automatically called at system initialization.
0	10	Disables the software graphics cursor. Use SET_CURSOR instead.
0	11	Enables the cursor. Use SET_CURSOR instead.
0	12	Makes the cursor visible. Use SET_CURSOR instead.
0	13	Makes the cursor invisible. Use SET_CURSOR instead.
0	14	Sets the cursor size to 15 pixels. Use SET_CURSOR instead.
0	15	Sets cursor to full screen. Use SET_CURSOR instead.
4	xx	Sets (xx) planes enables. Use PLANE_ENABLE instead.
7	xx	Erases screen to (xx) color. Use ERASE instead.
9	xx	Sets graphic cursor size to (xx) pixels. Use SIZE_CURSOR instead.

Effect on CP:

None.

Example:

SET_VALUE (7,3);

This example will erase the screen to color 3. The equivalent standard sequence is

CORE.Background :=3;

ERASE;

PROCEDURE ERASE

Declaration:

Procedure Erase;

Description:

This procedure erases the currently enabled planes to the background color. The background color is specified through the variable CORE .Background Index.

Effect on CP:

None.

Example:

CORE .Background_Index := 1

ERASE;

This example clears the screen and sets it to color 1.

PROCEDURE ERASE_ALPHA

Declaration:

Procedure Erase_Alpha:

Description:

This procedure erases the alphanumeric portion of the display. It leaves all graphics intact.

Effect on CP:

None.

Example:

ERASE_ALPHA

PROCEDURE FLOOD

Declaration:

Procedure Flood;

Description:

This procedure does an area fill with the color index specified in the CORE .Fill_Index originating from the CP. The area file operation is as follows:

1. The color at the current position is recorded as the base color.
2. Filling then occurs in all directions until a border is encountered.
3. A border is defined as pixel in any color other than the base color.

At this time the display mode (CORE .Display_Mode) must be set to Fast. In a future release, the ability to "flood" with a user-defined pattern will be available.

Effect on CP:

None.

Example:

```
CORE .Fill_Index := l;
Move_Abs( 30 , 20 );
Flood;
```

This example will fill the area around the point (30,20) with red.

PROCEDURE ARC_REL

Declaration:

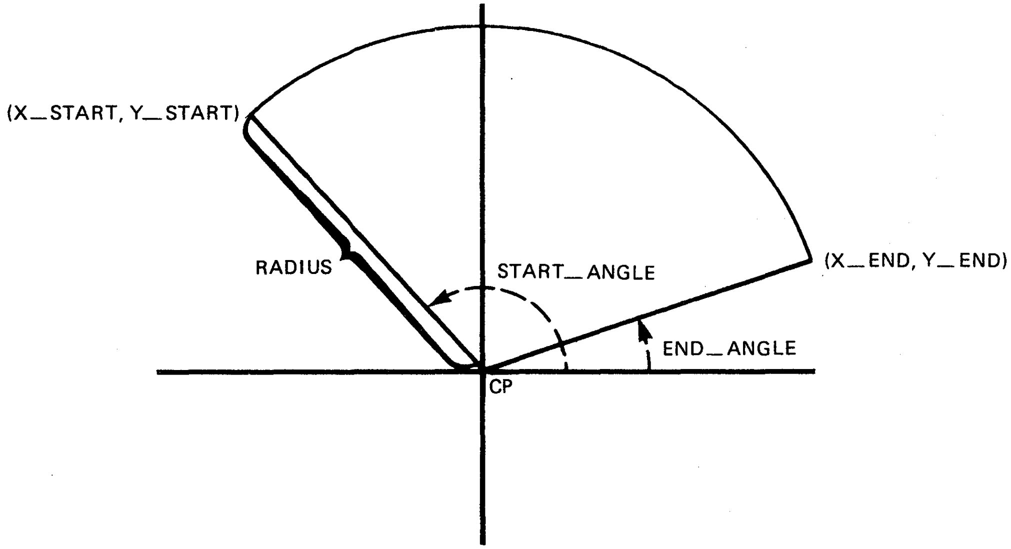
```
Procedure Arc_Rel (Radius       :Integer;
                   Start_Angle
                   End_Angle       :Real;
              Var  X_Start
                   Y_Start,
                   X_End
                   Y_End
```

Description:

This procedure draws part of a circle in the following manner:

1. The center of the circle is defined by the CP.
2. The radius of the circle to be drawn is passed to the procedure in the parameter Radius. It is measured in terms of pixels.
3. Two lines are then drawn of length Radius starting from the CP in the direction of Start_Angle and End_Angle.
4. The Start_Angle and End_Angle are measured in radians from the X-axis in a counterclockwise direction (that is, a line pointing directly up has an angle measurement of PI/2).
5. The part of the circle originating at Start_Angle and going in a clockwise direction to End_Angle is then drawn.
6. The X and Y coordinates of the point represented by the intersection of the arc with the line drawn at the angle represented by Start_Angle are returned to the program in the variables X_Start and Y_Start, respectively.
7. The X and Y coordinates described by the intersection of the arc with the line drawn at the angle represented by End_Angle are returned to the program in the variables X_End and Y_End.

The following illustrates the ARC_REL procedure.



Currently, the variable CORE . Display_Mode must be set to Fast. In a future release, it will be possible to fill an arc with a solid color or pattern.

Effect on CP:

None.

Example:

```
Var
    PI  :  Real;
        X_Start , Y_Start
        X_End , Y_END   :   Integer;

    PI := 3.14159;
    Move_Abs(75,100);
    CORE .Display_Mode: = Fast;
    CORE .Line_Index: = 2
    Arc_Rel(50,2*PI/3 , PI/6 , X_Start , X_End Y_End);
```

PROCEDURE ARC_ABS

Declaration:

```
Procedure Arc_Abs (Var Radius : Integer;
                   Var Start_Angle,
                       End Angle : Real;
                       X_Start,
                       Y_Start,
                       X_End,
                       Y_End     : Integer;
```

Description:

This procedure draws an arc in the same way as ARC_REL with the exception that X_Start , Y_Start , X_End, and Y_End must be passed as parameters, and Radius, Start_Angle, and End_Angle are returned by the procedure.

The arc is defined in the following manner:

1. The center of the circle is defined by the CP.
2. A line is drawn from the CP to the point defined by X_Start and Y_Start.
3. The length of this line is then returned to the calling program in the variable passed as Radius.

4. The angle at which the line was drawn (measured in the same way as described above), is returned in the variable passed to the procedure as Start_Angle.
5. A line is then drawn in the direction described by the parameters X_End and Y_End with a length equal to the length of the first line drawn (the value just placed in the variable Radius).
6. The arc is drawn at the angle at which this line was drawn starting from the angle just placed in Start_Angle, and continuing in a clockwise direction to the angle now described by End_Angle.

Currently, the variable CORE .Display_Mode must be set to Fast. In a future release, it will be possible to fill an arc with a solid color or pattern.

Effect on CP:

None.

Example:

```
Move_Abs(320,1000;
CORE .Display_Mode :=Fast;
CORE .Overlay_Mode:=Replace;
CORE .Line_Index:=2;
Arc_Abs(Radius , Start_Angle , End_Angle , 300 , 120 , 330 , 90 );
```

THE FONT COMPILER

The Font Compiler (FNTCOMP.EXE) accepts a series of text files containing a font definition and produces an .FNT data file suitable for use with the Graf_Draw unit.

To execute the Font Compiler, enter FNTCOMP. The compiler will prompt for the name of the first text file of the font definition. This file contains font parameters and the first part of the font definition.

The next prompt is for the name of the font data file. This should be specified as "FONTxx.FNT", where xx is a two-digit font number (for example, 00 or 15). The compiler will store the compiled font data in the file named.

The Font Compiler then processes the character definitions until the end of the text file is encountered. It then prompts for the name of a continuation text file. Supply that file's name, if there is one. If therc are no continuation files, pressing RETURN causes the Font Compiler to close the .FNT data file and terminate.

Font Text Files

Font text files are line oriented. The first line of the first text file in a font definition contains four numbers describing the font.

- The first number is the number of pixels in the horizontal direction.
- The second number is the number of pixels in the vertical direction.
- The third number is the ASCII value of the first character in the font definition (for example, 32 for space).
- The fourth number is the ASCII value of the last character in the font definition (for example, 127 for rubout).

The above numbers are separated by one or more spaces. The fourth number is followed by a carriage return code.

The remainder of the text file (and all of any continuation text files) contains character definitions, starting with the lowest valued character in the font and continuing without interruption to the highest valued character in the font.

A character definition consists of a line containing the character to be defined, enclosed in quotes, followed by several lines that define the way the character will be formed. Together, these several lines form a picture representing pixels that are on and off. Each line corresponds to one row of pixels in the character image. There are as many lines as there are rows in the character image (as specified by the second number described above, "number of pixels in the vertical direction"). Within a line, "."s represent pixels that are turned off, and other characters represent pixels that are turned on. Two spaces separate each "." or other character. There will be as many pixel characters on each line as specified by the first number described above, "number of pixels in the horizontal direction." An example of a character definition of an 8 by 14 pixel character is

```
"b"
. . . . . . . .
. b . . . . . .
. b . . . . . .
. b . . . . . .
. b . b b . . .
. b b . . b . .
. b . . . . b .
. b . . . . b .
. b . . . . b .
. b . . . . b .
. b b . . b . .
. b . b b . . .
. . . . . . . .
. . . . . . . .
```

Note that there is never any blank line within the font text file.

For a complete example of a font text file, see the FONT01.TXT file supplied with the supplement.

FONT DATA FILES

The format of the .FNT data file is

Word 1: Number of pixels in the horizontal direction
Word 2: Number of pixels in the vertical direction
Word 3: Value of the first character in the font
Word 4: Value of the last character in the font
Word 5-? Array [Word3..Word4] of character images.

Each character image is an array of byte-aligned rows. Each row occupies (Word1 + 7) Div 8 bytes. Each character occupies Word2 * ((Word1 + 7) Div 8) bytes. There are no padding bytes between rows. Character definitions are word aligned.

THE PATTERN COMPILER

The Pattern Compiler (PATCOMP.EXE) accepts a text file containing a fill pattern definition, and produces a .PTN file suitable for use with the Graf_Draw unit.

To execute the Pattern Compiler, enter PATCOMP. The compiler will prompt for the name of the text file containing the pattern definition. This file contains pattern parameters and the pattern definitions.

The next prompt is for the name of the pattern data output file to be produced. This should be specified as "PATxx.PTN", where xx is a two-digit pattern number (for example, 00 or 15). The compiler will store the pattern definition in the file named.

The Pattern Compiler processes the pattern definition until the end of the text file is encountered. It then closes the pattern data file and terminates.

Pattern text files are line-oriented. The first line of the text file contains two numbers describing the pattern. The first number is the number of pixels in the horizontal direction. The second number is the number of pixels in the vertical direction. The numbers are separated by one or more spaces. A carriage return follows the second number. The remainder of the text file contains the pattern definition.

A pattern definition consists of several lines containing a drawing consisting of color identifiers representing the value of each pixel separated by two spaces. The color identifiers are

D,d, or .	for dark (black)
R or r	for red
G or g	for green
Y or y	for yellow
B or b	for blue
P or p	for purple
T or t	for turquoise
W or w	for white

Each line corresponds to a row of the pattern image. There are as many lines and rows in the pattern as are specified by the numbers on the first line of the pattern text file.

And example pattern test file for an 11 by 11 pattern is

```
11 11
. b b b b b b b b b .
y . b b b b b b b . g
y y . b b b b b . g g
y y y . b b b . g g g
y y y y . b . g g g g
y y y y y . g g g g g
y y y y . r . g g g g
y y y . r r r . g g g
y y . r r r r r . g g
y . r r r r r r r . g
. r r r r r r r r r .
```

The format of the .PTN pattern data files produced by the compiler is

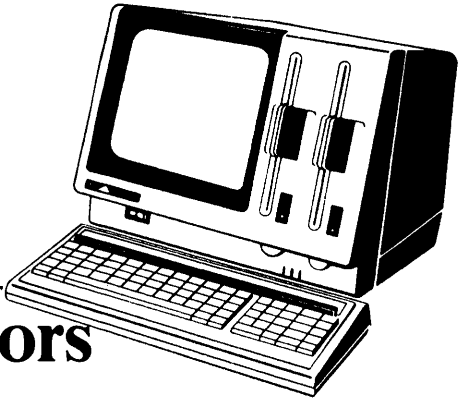
Word1:	number of pixels in the horizontal direction
Word2:	number of pixels in the vertical direction
Word 3-?:	Array [Word1..Word2] of pattern rows.

The pattern image is an array of word-aligned rows. Each row consists of an array of color identifiers each occupy four bits. A row occupies (Word1 + 3) Div 4 words. The pattern occupies Word 2 + ((Word1 + 3) Div 4) words. Note that this format corresponds to the internal representation of an array under the UCSD Pascal system. An array declaration for the example pattern might be

Array [1..11] of Packed Array [1..1] of 0..15;

The color token values for the possible colors are as follows:

Value	Color
0	for dark (black)
1	for red
2	for green
3	for yellow
4	for blue
5	for purple
6	for turquoise
7	for white



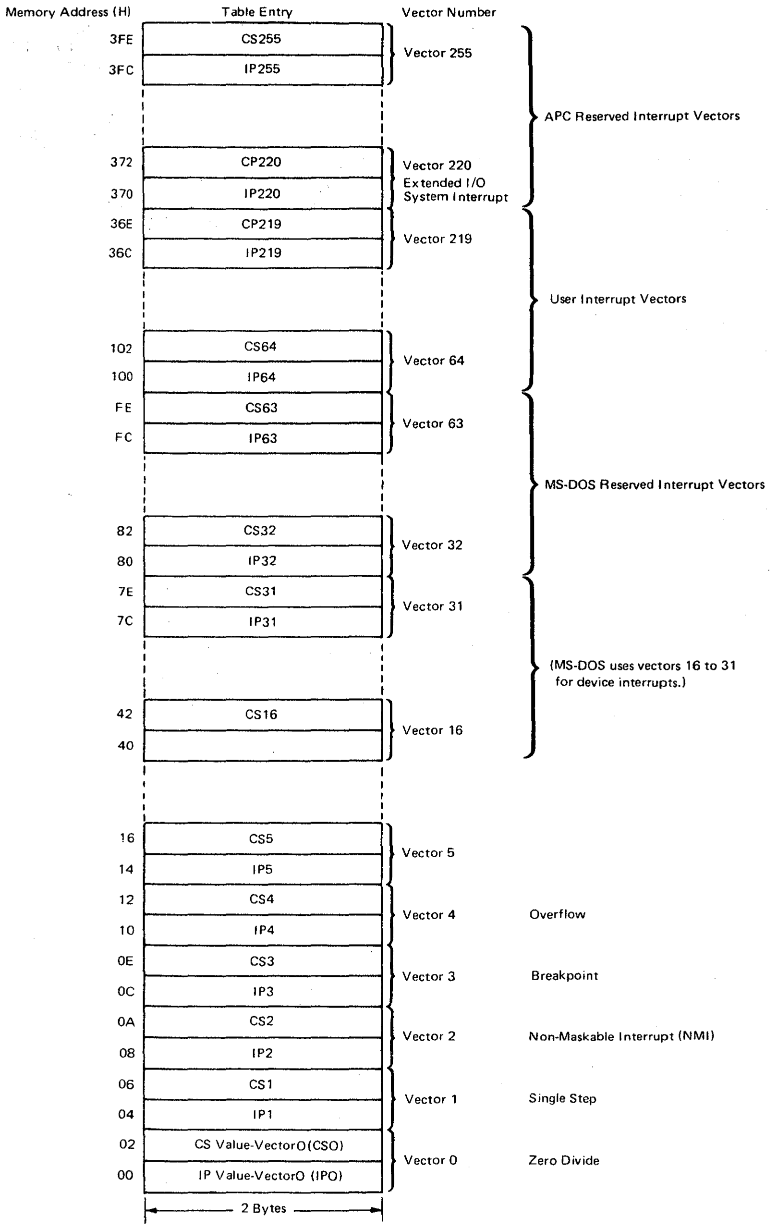
Appendix

The MS-DOS Interrupt Vectors

The MS-DOS interrupt vectors are as follows:

- CPU interrupt vectors
- Device interrupt vectors
- MS-DOS reserved interrupt vectors
- User interrupt vectors
- APC reserved interrupt vectors.

The interrupt vector table shown in Figure A-l consists of 256 entries. Each entry has two l6-bit address values (4 bytes), which are loaded into the code segment (CS) register and the instruction pointer (IP) register as the interrupt routine address when an interrupt occurs. This means absolute locations 0H to 3FFH are the transfer address storage locations.



Memory Address (H)	Table Entry	Vector Number	
3FE	CS255	Vector 255	APC Reserved Interrupt Vectors
3FC	IP255		
372	CP220	Vector 220 Extended I/O System Interrupt	
370	IP220		
36E	CP219	Vector 219	User Interrupt Vectors
36C	IP219		
102	CS64	Vector 64	
100	IP64		
FE	CS63	Vector 63	MS-DOS Reserved Interrupt Vectors
FC	IP63		
82	CS32	Vector 32	
80	IP32		
7E	CS31	Vector 31	(MS-DOS uses vectors 16 to 31 for device interrupts.)
7C	IP31		
42	CS16	Vector 16	
40			
16	CS5	Vector 5	
14	IP5		
12	CS4	Vector 4	Overflow
10	IP4		
0E	CS3	Vector 3	Breakpoint
0C	IP3		
0A	CS2	Vector 2	Non-Maskable Interrupt (NMI)
08	IP2		
06	CS1	Vector 1	Single Step
04	IP1		
02	CS Value-VectorO(CSO)	Vector 0	Zero Divide
00	IP Value-VectorO (IPO)		

Figure A-1 MS-DOS Interrupt Vector Table

CPU INTERRUPT VECTORS

There are two types of CPU interrupt: the software interrupt and the hardware interrupt. A hardware interrupt is classified as either a non-maskable interrupt (NMI) or maskable interrupt. Regardless of its type, an interrupt results in the transfer of control to a new location.

DEVICE INTERRUPT VECTORS

MS-DOS uses vectors 16 to 31 for device interrupts. This means absolute locations 40 to 7F hex are the transfer address storage locations used by IO.SYS. The interrupts are as follows:

Vector 16	All stop	(Not currently used.)
Vector 17	Communication	(Not currently used.)
Vector 18	Option	(Not currently used.)
Vector 19	Timer	
Vector 20	Keyboard	
Vector 21	Option	(Not currently used.)
Vector 22	Option	(Not currently used.)
Vector 23	ODA Printer	(Not currently used.)
Vector 24	Option	(Not currently used.)
Vector 25	Option	(Not currently used.)
Vector 26	CRT	(Not currently used.)
Vector 27	FDD	(Not currently used.)
Vector 28	Option	(Not currently used.)
Vector 29	Option	(Not currently used.)
Vector 30	APU	(Not currently used.)
Vector 31	Option	(Not currently used.)

MS-DOS RESERVED INTERRUPT VECTORS

MS-DOS reserves vectors 32 to 63 (absolute locations 80 to FF hex) for the DOS. These interrupts are as follows:

Vector 32 Program terminate. This is the normal way to exit a program. This vector transfers to the logic in the the DOS for restoration of CNTL-C exit addresses to the values they had on entry to the program.

Vector 33 Function request.

Vector 34 Terminate address. If a program is to execute a second program, it must use Terminate Address prior to creation of the segment into which the program will be loaded.

Vector 35 CNTL-C exit address.

Vector 36 Fatal error abort vector. When a fatal error occurs, control will be transferred with an INT 24H.

Vector 37 Absolute disk read.

Vector 38 Absolute disk write.

Vector 39 Terminate but stay resident. This vector is used by programs that are to remain resident when COMMAND.COM regains control.

USER INTERRUPT VECTORS

MS-DOS allows you to use vectors 64 to 219. These vector's values are initialized to invoke an interrupt fault process in IO.SYS. If you use any of them, you must set its value. Be sure to reset the vector to the initial value when you have completed your task.

APC RESERVED INTERRUPT VECTORS

MS-DOS reserves the vectors 220 to 256 (absolute locations 370 to 3FF hex) as the transfer address storage locations for the APC extended functions. The one interrupt currently defined is vector 220 for extended function call entry.

Advanced
Personal Computer

NEC
NEC Information Systems, Inc.

USER'S COMMENTS FORM

Document: MS™-DOS System Reference Guide

Document No.: 819-000103-2001 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. 386 LEXINGTON, 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.